万水计算机编程技术与应用系列

注：

本书程序样例源代码可到中国水利水电出版社网站上下载
(www.waterpub.com.cn)

OpenGL 图形程序设计指南

尚游　陈岩涛　编著

中国水利水电出版社
www.waterpub.com.cn

责任编辑：轩志勇　封面设计：凤　歌

万水计算机编程技术与应用系列

- 《Delphi 5.0 编程技巧与实例分析》（含光盘）
- 《ASP 编程技巧与实例分析》（含光盘）
- 《PHP 编程技巧与实例分析》（含光盘）
- 《Perl 5 编程技巧与实例分析》（含光盘）
- 《OpenGL 图形程序设计指南》
- 《Visual Basic 6.0 编程技巧与实例分析》（含光盘）
- 《Visual FoxPro 6.0 编程技巧与实例分析》（含光盘）
- 《Visual C++ 6.0 编程技巧与实例分析》（含光盘）

ISBN 7-5084-0839-X
9 787508 408392 >

北京万水电子信息有限公司
Beijing Multi-Channel Electronic Information Co., Ltd.
地址：北京市西直门外榆树馆一巷永康商务写字楼
邮编：100044
电话：(010)6835.9286　6835.9167
传真：(010)6835.9284
E-mail: mchannel@public3.bta.net.cn

ISBN 7-5084-0839-X/ TP · 309
定价：48.00 元

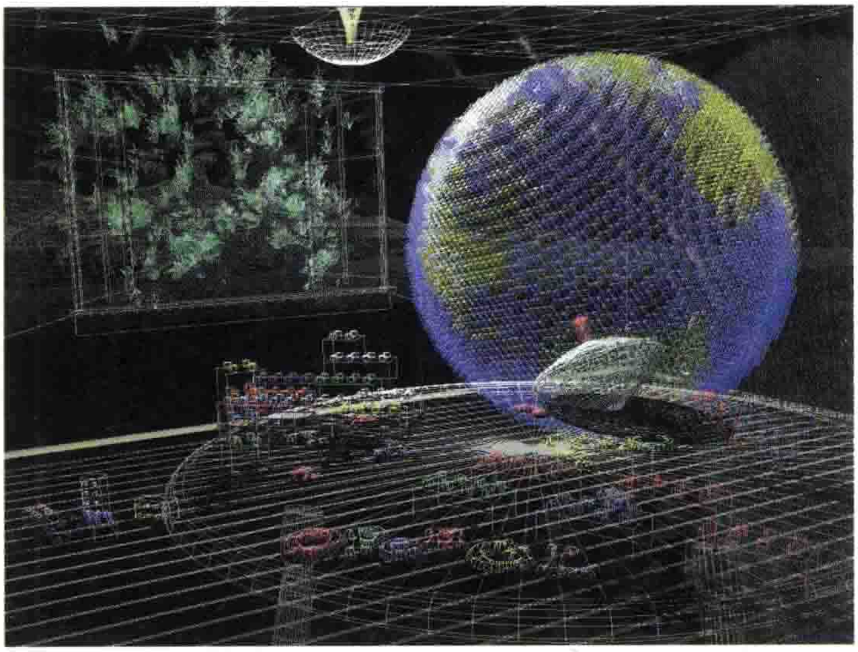
万水计算机编程技术与应用系列

OpenGL 图形程序设计指南

尚　游　陈岩涛　编著

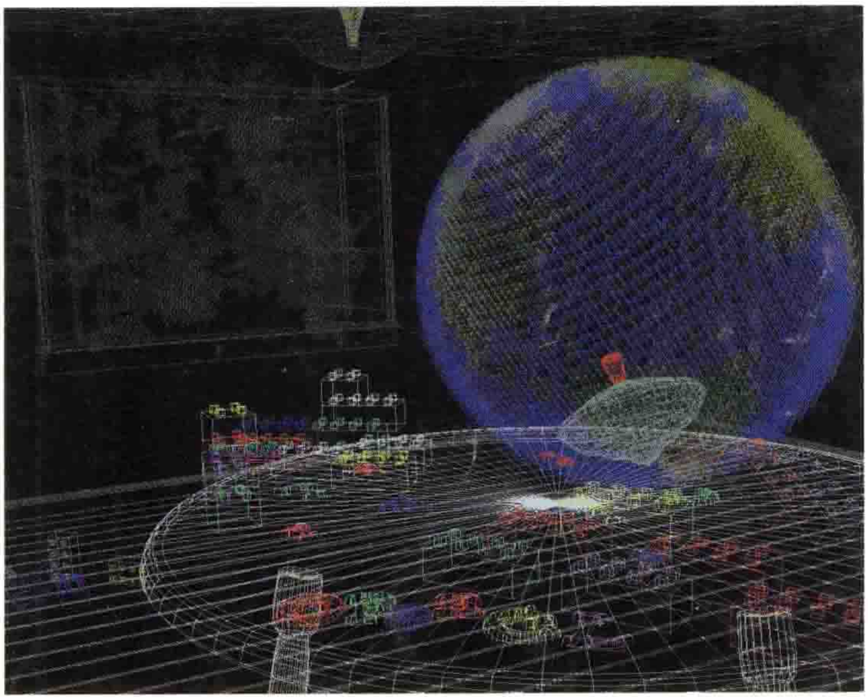
中国水利水电出版社

OpenGL 应用程序生成的场景彩图



彩图 1

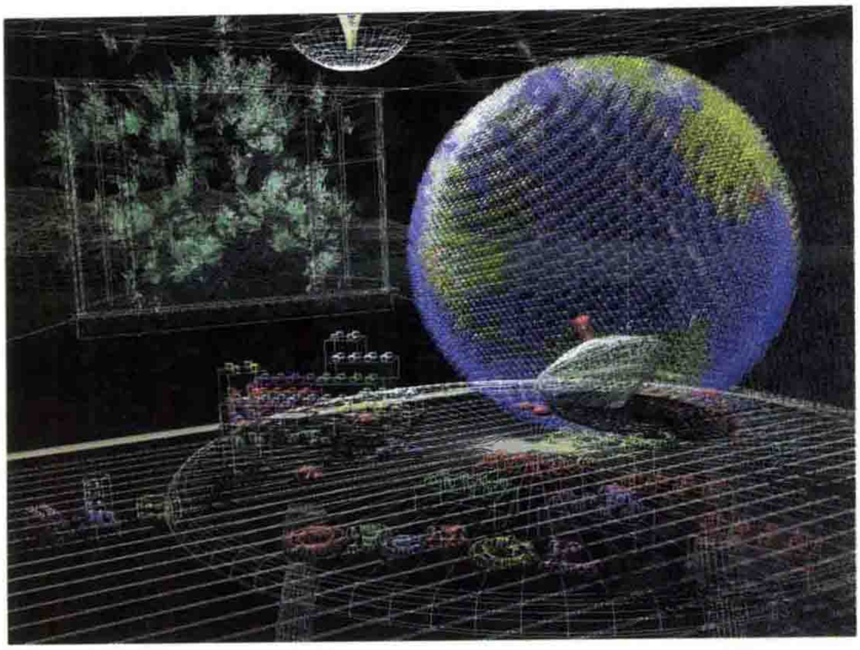
彩图 1 是由 OpenGL 生成的物体的线框图。参见第 3 章。



彩图 2

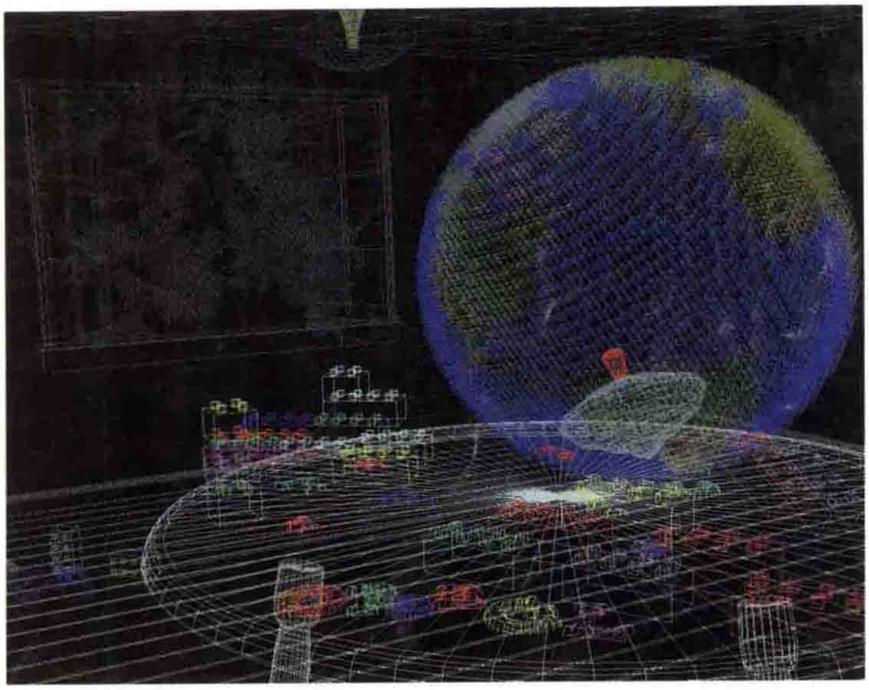
彩图 2 与彩图 1 是相同的场景，用雾来实现深度提示（离眼睛越远的线条显得越暗）效应。参见第 8 章。

OpenGL 应用程序生成的场景彩图



彩图 1

彩图 1 是由 OpenGL 生成的物体的线框图。参见第 3 章。



彩图 2

彩图 2 与彩图 1 是相同的场景，用雾来实现深度提示（离眼睛越远的线条显得越暗）效应。参见第 8 章。

彩图 6

彩图 6 加入了纹理图和阴影。参见第 10 章和第 14 章。

彩图 7

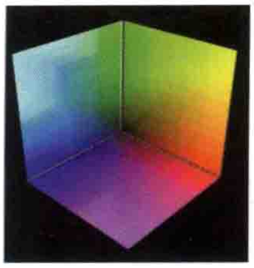
彩图 7 中的物体具有运动模糊效果。参见第 11 章。

彩图 10

彩图 10 绘制了一些具有景深效果的茶壶。参见第 11 章。

彩图 11

彩图 11 是具有平滑明暗效果的三角形。参见第 6 章。



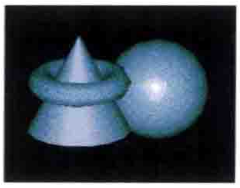
彩图 12

彩图 13

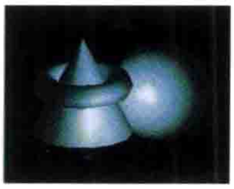
彩图 12 和 13 为着色的立方体。彩图 12 的轴分别为红色、绿色和蓝色；彩图 13 的轴分别为黄色、青色和洋红色。参见第 6 章。

彩图 14 a

彩图 14 b



彩图 14 c



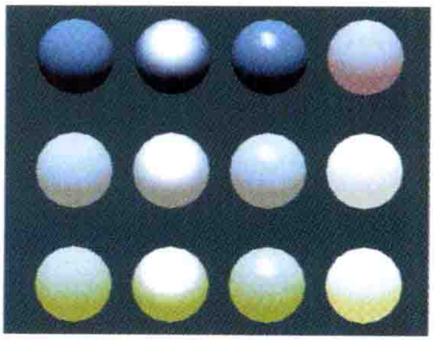
彩图 14d

彩图 14是以灰色材质参数和彩色光源绘制的物体。(a) 场景有淡蓝色的环境光和白色散射光源。(b) 场景有淡蓝色的散射光源，但几乎没有环境光。(c) 场景使用了平行光源。(d) 使用局部光源。参见第 7 章。

彩图 15a

彩图 15b

彩图15是用不同光照条件绘制的灰色茶壶。(a) 三个茶壶依次以渐强的环境光绘出。(b) 剪裁茶壶，将内部暴露出来。最上面的茶壶用的是单侧光，中间的用双侧光，正面和背面用同样的材质，最下面的茶壶用双侧光且正面和背面材质不同。参见第 7 章。



彩图 16

彩图16是十二个材质参数各不相同的球体。各行的性能：第 1 行没有环境反射；第 2 行是灰色环境反射；第 3 行为蓝色环境反射。第一列使用了没有镜面反射属性的蓝色散射材质颜色。第二列加入了亮度指数较低的白色镜面反射。第三列加入了较高的亮度指数，因此高亮区更集中。第四列使用了蓝色散射颜色，用辐射指数代替镜面反射。参见第 7 章。

彩图 22a

彩图 22b

彩图 23a

彩图 23b

彩图 22 是几个明暗效果的物体。(a) 有锯齿现象。(b) 消除了锯齿现象。参见第 10 章。

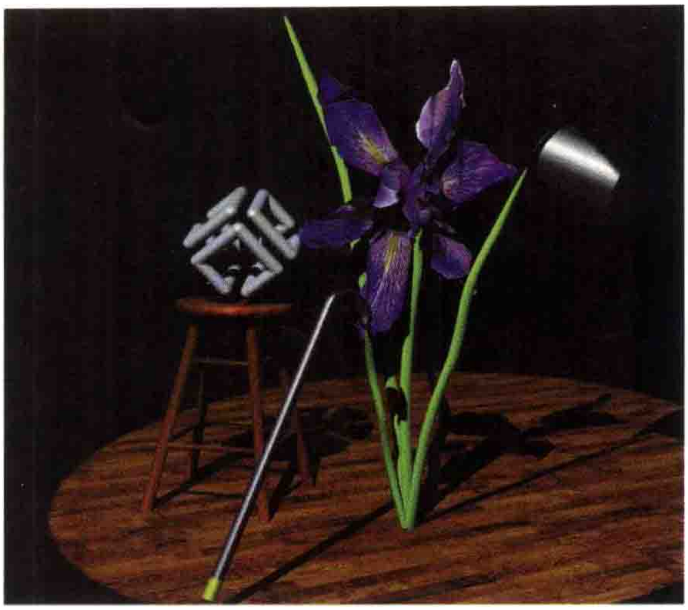
彩图 23 是前面场景的放大。(a) 显示出锯齿状的边缘。(b) 是消除了锯齿现象的边缘。参看第 11 章。

彩图 24

彩图 24 是用纹理映射、光照和阴影绘制的场景。参见第 7 章、第 10 章和第 14 章。

彩图 25

彩图 25 是在纹理表面上的光照明暗效果平滑的一个模型。参见第 6 章，第 7 章和第 10 章。



彩图 26

彩图 26 是有明显光照和阴影效果的场景，大部分表面都有纹理。参见第 3 章、第 7 章和第 14 章。

彩图 27

彩图 28

彩图 27 和 28 是由可视化仿真程序生成的两幅场景。参见第 4 章，第 5 章和第 11 章。

彩图 29

彩图 30

彩图 29 为利用 OpenGL 制作的动画屏幕保护程序，参见第 15 章。

彩图 30 为利用分形算法生成的三维地形及蓝天白云图，参见第 16 章。

彩图 17

彩图17是以接近真实材料的不同材质绘制的有光照的明暗效果平滑的茶壶。参见第7章。

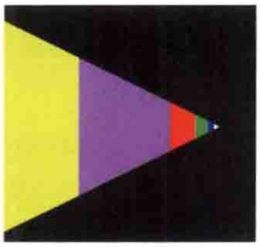
彩图 18a

彩图 18b

彩图 18c

彩图18是使用自动纹理坐标和红色轮廓纹理图生成的有光照的绿色茶壶。(a)纹理轮廓条纹平行于平面x = 0。(b) 平面方程为 (x + y + z = 0)，因此条纹具有不同方向。(c) 相对于人眼坐标计算的纹理坐标，纹理并不固定在物体上。因此当物体运动时，看起来就像是在纹理中游动。参见第 10 章。

彩图 19



彩图 20

彩图 21a

彩图 21b

彩图 19 是用求值程序创建的有纹理的 Bezier 曲面网格。参见第 10 章和第 12 章。

彩图 20 用一系列 mipmaps 纹理绘制出一个多边形。在本例中，每个纹理仅仅只有颜色不同。当多边形的可见区域变小时，相应地使用较小的 mipmaps。参见第 10 章。

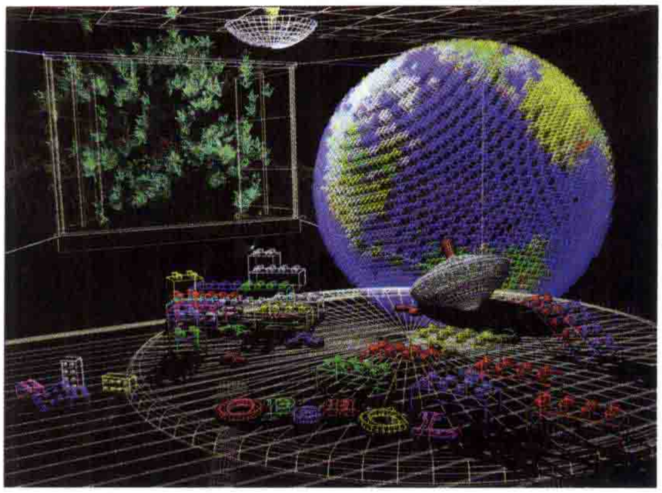
彩图 21 为反射环境的物体。(a) 为原始纹理，是由广角镜拍摄的处理过的咖啡店的照片。(b) 是应用该纹理图的高脚酒杯。由于映射作用，高脚酒杯表面看起来反射出了咖啡店情景。

彩图 8

彩图 8 为场景的特写镜头。参见第 4 章。

彩图 9

彩图 9 利用大气效果（雾）模拟充满烟雾的房间。参见第 7 章。



彩图 3

彩图 3 也是同样一幅场景，但进行了反走样操作使边缘变得平滑。参见第 7 章。

彩图 4

彩图 4 是用具有无明暗效果的多边形绘出的场景。参见第 6 章。

彩图 5

彩图 5 是用光照和平滑明暗效果的多边形渲染的场景。参见第 5 章和第 6 章。

前　言

简介

图形处理是现代计算机区别于以前计算机的一个显著特征，使计算机走出了呆板的单一枯燥计算，在保证完成任务的同时更加接近真实世界，从而大大扩展了计算机应用领域，并带给编程人员和应用者丰富多彩的视听感受。

OpenGL 是 SGI 公司开发的一套高性能的图形处理系统，是图形硬件的软件界面，GL 即代表图形库（Graphics Library）。通过 OpenGL 程序员可以创建交互式应用程序，实现具有逼真效果的三维图形图像，从而在要求高度模拟真实世界的诸多领域中都可以大显身手。

与 AutoCAD，3DS MAX 等通用绘图及建模软件平台不同，OpenGL 不仅可以处理单幅的离散图形，在实时的三维仿真领域，由于其对环境及实体的渲染达到了高度逼真的视觉效果，从而显示出强大的生命力。

本书根据编者的实际应用经验，参照大量的原始资料，通过大量的实例详尽地阐述利用 OpenGL 进行图形程序设计的方法及技巧。

广阔的应用背景

应用程序可视化是计算机程序设计的发展趋势。在工程应用的许多领域中，计算机仿真的蓬勃发展使得人们可以不必在真实试验场便可以获得许多系统模型的性能，即使是真实试验，也少不了利用仿真系统对试验的再现。计算机仿真已经成为工程设计和软件开发人员不可或缺的重要工具。

涉及图形图像处理的应用领域，尤其是要求高实时性、大计算量和渲染量的仿真中，如：

- 动力学系统的可视化仿真。
- 结构应力有限元分析。
- 人体医学扫描数据显示。
- 交互式电子游戏开发。
- 三维造型设计。
- 动画制作。
- 广告设计。

这些都是 OpenGL 图形库的用武之地。不难看出，上述各个领域都是现代计算机应用的热门方向。图形处理方兴未艾，随着要求不断提高，竞争不断加剧，OpenGL 图形库必然占据不久以后的图形处理领域。

虽然 OpenGL 目前主要应用于图形工作站中的软件平台如 IRIS Performer，但是越来越

多的软件开发商们已经注意到了 OpenGL 图形库在微机上所具备的潜在前景。包括微软在内的许多著名公司所提供的编译平台，如 Visual C++ 6.0、C++ Builder、VB 等开发工具中，都将 OpenGL 作为一个重要的模块扩展进来，从而大大增强该编译平台对图形的处理能力。经典的三维造型设计软件 3DS MAX 在 R2 以上版本中也提供了与 OpenGL 图形库的接口。这不但从一个侧面表明了广大开发商对 OpenGL 图形库强大能力和潜在市场的认可，同时也证明了它的良好的可移植性。

本书的主要特色

OpenGL 是一种功能强大，应用与开发前景广阔的软件平台，顺应着计算机工程应用可视化的潮流。本书的目的是要向设计及开发工程师提供 OpenGL 的全面和系统的操作指南和参考资料。本书具有以下几个特色。

1. 丰富、完备的参考资料

本书主要的参考资料来自 SGI 工作站上的原文在线帮助、编程指南和库函数说明，辅助的参考材料为 Visual C++ 6.0 及 C++ Builder 的 OpenGL 帮助和库函数说明，本书包括的源代码全部经过作者的实际调试，部分程序由作者开发，具有很强的实用性。

2. 将应用扩展到 PC 机

个人计算机计算速度的大幅度提高使得它们与工作站的差距在不断缩小。虽然在相当长的一段时间内还不能期望其与专门从事图形处理的 SGI 工作站相比，但是已经具备运行 OpenGL 应用程序的能力，软件商们也不甘示弱，各个软件编译平台都将 OpenGL 模块拓展集成到自己的麾下，使得熟悉 C、C++等语言及其相应编译器 VC++、C++ Builder 的软件工程师可以在 586 以上的微机上编制非常 COOL 的 OpenGL 程序。OpenGL 已经渐渐走向成熟期。在本书的编写过程中，作者将所有的源代码都移植到了微机环境下。在源代码中有关于海洋生物的仿真程序，该程序涉及到 OpenGL 的建模、纹理、光照及渲染等多种高级技术，对读者来说是一个不可多得的入门程序。

3. 渲染平台与建模平台的互相扩展

与其他图形处理软件如 AutoCAD、3DS MAX 等相比，OpenGL 最强大的优势在于其实时渲染复杂场景的能力。如果它能够与广泛使用着的建模平台相结合起来，这将使 OpenGL 在不久的将来迅速地占领图形处理领域。针对 AutoCAD 的 DXF 文件，作者对 OpenGL 与 AutoCAD 的文件接口进行了探索性的研究，并取得了一定的成绩。本书的第 15 章介绍了 OpenGL 与 AutoCAD 之间的接口及实现。

本书的主要内容安排

全书包括 16 章正文及 10 个附录，各章的安排如下：

第 1 章：OpenGL 入门。本章概括地介绍了 OpenGL 的基本原理和基本功能，以及 OpenGL 的各种应用领域。然后通过一个简单的例程，解释了利用 OpenGL 编写程序的基本

要领。本章的内容是学习后续章节的基础。

第 2 章：Windows 95 及 Windows NT 环境下开发 OpenGL 应用程序的基本过程。本章内容着重介绍了通用微型计算机操作系统下的 OpenGL 图形程序的编制开发过程，引导读者练习编译运行本书的例程，为进一步开发自己的应用程序做好准备。

第 3 章：实体建模基础。三维几何体最终要绘制到二维屏幕上。复杂的三维物体是由基本的几何图元构成的。这部分内容主要介绍如何对三维物体进行几何上的描述，以及通过程序创建三维几何物体。

第 4 章：视图。本章主要介绍了在将三维的物体绘制到二维的屏幕上之前，如何对所创建的三维模型实施各种变换。程序员可以通过对这些变换进行控制，来显示模型的特定视图。

第 5 章：显示列表。本章讨论了如何缓存多条 OpenGL 命令，以及如何实现 OpenGL 命令的批处理。程序开发人员可以利用这种特性来提高 OpenGL 应用程序的执行性能。

第 6 章：颜色。本章描述了在绘制物体时如何指定物体的颜色，以及如何生成效果逼真的阴影的方法。

第 7 章：光照。本章解释了如何控制物体周围的光照条件，以及物体对光照的响应（如吸收或反射等）。光照是 OpenGL 中重要的一个主题。如果没有光照，物体将缺乏真实的三维立体感。

本书的其余各章介绍了如何将更加复杂的特性加入到三维场景中，以及 OpenGL 作为一种渲染工具对建模工具中的实体模型的调用。在对 OpenGL 不是很熟悉时，读者可以暂不使用这些特性。下面列出了介绍这些特殊的高级技术的相关章节的主要内容。

第 8 章：混合、反走样、雾。本章描述了创建效果逼真的场景所必需的一种高级技术：alpha 混合（使程序员能创建透明物体），锯齿现象的消除及大气效果（如雾或烟等）。

第 9 章：绘制像素、位图、字体和图像。本章讨论如何将二维数据集合（如位图或图像）与三维物体的操作结合起来。位图的典型应用是用字体来描述字符。

第 10 章：纹理映射。本章解释了如何将一维或二维的称为纹理的图像映射到三维物体上。利用纹理映射可以产生许多奇妙的效果。

第 11 章：帧缓存。本章描述了在 OpenGL 的实现过程中可能用到的所有缓存，以及如何控制这些缓存。程序员可以利用这些缓存实现下列效果：隐藏面的消除，模板化，屏蔽，运动模糊化以及视野的聚焦等等。

第 12 章：求值程序和非均匀有理 B 样条（NURBS）。本章介绍了有效地生成曲线和曲面的高级技术。

第 13 章：选择机制。本章解释了如何利用 OpenGL 的选择机制来选中屏幕上的物体，同时解释了反馈机制，这种操作使程序员不需经过屏幕作图就可以采集到 OpenGL 生成的绘图信息。

第 14 章：OpenGL 应用及提高应用程序性能的技巧。本章描述了如何以巧妙的、出人意料的方式，利用 OpenGL 生成有趣的结果。这些技术其中有些是 SGI 的技术先驱们经过

多年的经验积累得来的。当然，这里面更有作者长期编程实践的体会。

第 15 章：高级应用（一）：OpenGL 与 AutoCAD 的图形接口。OpenGL 作为一种功能强大的渲染工具，与通用的建模工具结合起来，各取其长，将同时获得好的性能和效果。本章分析了 DXF 文件的数据结构，描述了在 OpenGL 中调用 DXF 文件的通用函数模块，同时给出了调用函数的程序源代码。

第 16 章：高级应用（二）：三维随机分形地形的生成。OpenGL 可以实现许多复杂的场景渲染。在这一章中，作者利用数学上的分形原理来生成复杂的三维地形及天空的模拟，同时给出了一个完整的三维地形模拟源程序。相信读者在学习完这一章之后，对 OpenGL 的强大功能会有更深刻的体会。

另外，我们在本书中列出了 10 个附录。这些附录对帮助程序员理解 OpenGL 的原理及机制，以及编写高效的应用程序是不无裨益的，其中对库函数的介绍部分是以应用类别排列，与库函数详解相区别。

附录 1：OpenGL 中的操作顺序。本附录给出了 OpenGL 应用操作的技术概览，并按照这些技术在应用程序中执行的顺序对它们进行了简洁的描述。

附录 2：OpenGL 的状态变量。本附录列出了 OpenGL 所支持的状态变量，并描述了如何获取这些变量的值。

附录 3：OpenGL 实用库。本附录简洁地描述了在 OpenGL 实用库中的实用子程序。

附录 4：OpenGL 对 X 窗口系统的扩展。本附录简要描述了在 OpenGL 对 X 窗口系统扩展中，程序员可以使用的子程序。

附录 5：OpenGL 辅助库。本附录简要介绍 OpenGL 的辅助库（GLAUX）。

附录 6：计算法线向量。本附录介绍了几种不同类型的几何体的法线向量的计算方法。

附录 7：齐次坐标和变换矩阵。本附录介绍了矩阵变换所需的基本的数学知识。

附录 8：OpenGL 编程要点。本附录列出了一些 OpenGL 编程要点。这些要点是从 OpenGL 的设计者的角度来考虑的，但在编制程序时读者将会发现这些要点是非常有用的。

附录 9：OpenGL 的不变性。描述了各种 OpenGL 实现所遵循的基本不变性规律。

附录 10：术语表。介绍了各种 OpenGL 编程常用术语的定义。

预备知识与参考书目

阅读和学习本书之前，需要读者了解如何运用 C/C++语言编程，并具有计算机图形学方面的初步知识（如几何学，三角法，线性代数，微积分学及微分几何学）。当然，计算机图形技术的经验很少甚至没有的人，仍能够参照使用本书中大部分内容。计算机图形学是一个非常丰富的领域，为丰富学习经验，读者可以参阅国内外相关的参考资料，如计算机图形学，Visual C++编程，OpenGL 程序设计等。

目　　录

前言

第 1 章　OpenGL 入门 1

1.1　OpenGL 概览 1

1.2　简单的 OpenGL 程序举例 3

1.3　基本 OpenGL 操作 5

1.4　OpenGL 的命令语法 6

1.5　作为状态机的 OpenGL 7

1.6　OpenGL 相关函数库 8

1.6.1　OpenGL 相关函数库简介 8

1.6.2　OpenGL 辅助库（GLAUX） 9

1.7　动画 11

第 2 章　Windows 95 及 Windows NT 环境下开发 OpenGL 应用程序的基本过程 17

2.1　开发环境 17

2.1.1　系统硬件配置要求 17

2.1.2　开发 OpenGL 应用程序的软件要求 17

2.2　OpenGL 应用程序开发基本过程 18

2.2.1　源代码的编写 18

2.2.2　Win32 控制台风格应用程序的开发 18

2.2.3　Win32 MFC 风格应用程序的开发 22

第 3 章　实体建模基础 24

3.1　一个绘图补救工具包 24

3.1.1　清除窗口 25

3.1.2　指定颜色 27

3.1.3　完成绘图 28

3.1.4　隐藏面的消除 29

3.2　点、线和多边形的描述 30

3.2.1　点、线和多边形的定义 30

3.2.2　定义顶点 33

3.2.3　OpenGL 的几何绘图图元 34

3.2.4　使用 glBegin()和 glEnd()的限制 37

3.3 显示点、线和多边形 38
3.3.1 关于点的细节 38
3.3.2 关于线的细节 39
3.3.3 关于多边形的细节 43
3.4 法线向量 49
3.5 创建表面的多边形模型需注意的问题 51
第 4 章 视图 59
4.1 概述：照相机类比法 60
4.1.1 照相机类比法及其在计算机中的实现 60
4.1.2 一个简单的例子：绘制立方体 62
4.1.3 多种用途的变换函数 66
4.2 视图变换和模型变换 67
4.2.1 关于变换的说明 68
4.2.2 模型变换 70
4.2.3 视图变换 74
4.3 投影变换 78
4.3.1 透视投影 78
4.3.2 正交投影 80
4.3.3 视图体的裁剪 82
4.4 视区变换 82
4.4.1 定义视区 82
4.4.2 变换后的 Z 坐标 84
4.5 变换的诊断 84
4.6 矩阵堆栈的操作 86
4.6.1 模型视图矩阵堆栈 88
4.6.2 投影矩阵堆栈 88
4.7 附加裁剪平面 89
4.8 组合变换举例 92
4.8.1 建立一个太阳系模型 92
4.8.2 建立一个有关节的机器人手臂模型 95
第 5 章 显示列表 100
5.1 一个使用显示列表的范例 100
5.2 显示列表设计准则 102
5.3 显示列表的创建和执行 104
5.3.1 显示列表的存储内容 106

5.3.2 执行显示列表 107
5.3.3 显示列表的层次结构 109
5.4 显示列表及其索引的管理 110
5.5 执行多个显示列表 111
5.6 封装模式的改变 117
第 6 章 颜色 119
6.1 颜色的感知 119
6.2 计算机的颜色 120
6.3 RGBA 模式与颜色索引模式 122
6.3.1 RGBA 模式与颜色索引模式的对比 122
6.3.2 RGBA 显示模式 123
6.3.3 抖动操作 123
6.3.4 颜色索引显示模式 125
6.3.5 在 RGBA 模式与颜色索引模式之间进行选择 126
6.4 指定颜色和阴影模型 126
6.4.1 在 RGBA 模式下指定颜色 127
6.4.2 在指定颜色索引模式下指定颜色 128
6.4.3 指定阴影模型 128
第 7 章 光照 132
7.1 真实世界与 OpenGL 光照 133
7.1.1 发射光，环境光，散射光和镜面反射光 133
7.1.2 材料颜色 134
7.1.3 光线与材质的 RGB 值 134
7.2 一个简单的例子：光照球体的渲染 135
7.2.1 为所有的物体顶点定义法线向量 137
7.2.2 创建，定位和激活一个或多个光源 137
7.2.3 选择光照模型 137
7.2.4 定义场景中物体的材质属性 137
7.2.5 一些重要说明 138
7.3 创建光源 138
7.3.1 颜色 139
7.3.2 定位和衰减 140
7.3.3 聚光灯 140
7.3.4 多光源 141
7.3.5 控制光源的位置与方向 142

7.4 选择光照模型 147
7.4.1 全局环境光 147
7.4.2 局部或无穷远的视点 147
7.4.3 双面光照 148
7.4.4 激活光照 148
7.5 材料属性的定义 148
7.5.1 散射和环境光反射 150
7.5.2 镜面反射 150
7.5.3 发射光颜色 151
7.5.4 改变材质属性 151
7.6 光照的数学计算 156
7.6.1 材质的发射光 157
7.6.2 放大的全局环境光 157
7.6.3 光源的贡献 157
7.6.4 聚光灯效果 157
7.6.5 环境光项 158
7.6.6 散射光项 158
7.6.7 镜面反射光项 158
7.6.8 求和 159
7.6.9 颜色索引模式下的光照 159
7.6.10 颜色索引模式下光照的数学计算 160
第 8 章 混合、反走样、雾 161
8.1 混合 161
8.1.1 源因子和目标因子 162
8.1.2 混合的简单使用 163
8.1.3 混合操作范例 164
8.1.4 具有深度缓存的三维混合 167
8.2 反走样 171
8.2.1 点和线的反走样 173
8.2.2 多边形的反走样 178
8.3 雾 182
8.3.1 雾的使用 182
8.3.2 雾的方程 186
第 9 章 绘制像素、位图、字体和图像 191
9.1 位图和字体 191

9.1.1 当前光栅位置 194
9.1.2 绘制位图 195
9.1.3 字体和显示列表 196
9.1.4 一个完整的字体定义和使用 197
9.2 图像 203
9.2.1 像素数据的读、写和拷贝 203
9.2.2 图像的放大与缩小 206
9.3 像素的存储、变换和映射 206
9.3.1 像素模式概论 206
9.3.2 像素存储控制模式 207
9.3.3 像素传递操作 209
9.3.4 像素矩形 210
9.3.5 像素读取过程详解 212
9.3.6 像素映射 213
第 10 章 纹理映射 215
10.1 概述及范例 217
10.1.1 纹理映射的步骤 217
10.1.2 一个简单的程序 218
10.2 指定纹理 221
10.2.1 使用纹理边界 222
10.2.2 细节等级（Level of Details） 222
10.2.3 控制过滤 228
10.2.4 一维纹理 230
10.3 调节和混合 230
10.4 分配纹理坐标 232
10.4.1 计算纹理坐标 232
10.4.2 纹理的重复和截取 233
10.5 自动纹理坐标生成 235
10.5.1 创建轮廓 236
10.5.2 环境映射 239
10.6 高级特性 240
10.6.1 纹理矩阵堆栈 240
10.6.2 纹理的 q 坐标 241
第 11 章 帧缓存 242
11.1 缓存及其使用 243

11.1.1 各种缓存概述 244
11.1.2 清空缓存 245
11.1.3 为写操作选择颜色缓存 246
11.1.4 缓存屏蔽 246
11.2 片原的测试与操作 247
11.2.1 剪取测试 248
11.2.2 alpha 测试 248
11.2.3 模板测试 249
11.2.4 深度测试 254
11.2.5 混合，抖动及逻辑操作 255
11.3 累积缓存 257
11.3.1 场景反走样 257
11.3.2 运动模糊度 269
11.3.3 景深 270
11.3.4 柔和阴影 276
11.3.5 抖动 276
第 12 章 求值程序和非均匀有理 B 样条（NURBS） 278
12.1 概述 278
12.2 求值程序 279
12.2.1 一维求值程序 279
12.2.2 二维求值程序 284
12.3 GLU 的 NURBS 接口程序 296
12.3.1 简单的 NURBS 范例 296
12.3.2 修剪 302
第 13 章 选择机制 306
13.1 选择 306
13.1.1 基本步骤 307
13.1.2 创建命名堆栈 307
13.1.3 选中记录 309
13.1.4 一个选择的实例 309
13.1.5 拾取 314
13.1.6 用多个命名和分层模型进行拾取 320
13.1.7 拾取和深度值 322
13.1.8 在编程中使用选择机制的一些提示 326
13.2 反馈 327

13.2.1 反馈数组 328
13.2.2 在反馈模式下使用标记 329
13.2.3 一个反馈的例子 330
第 14 章 OpenGL 应用及提高应用程序性能的技巧 335
14.1 轻松制作透明物体 336
14.2 轻松实现淡出效果 336
14.3 利用后缓存进行物体选择 337
14.4 方便的图像变换 338
14.5 分层显示 339
14.6 字符反走样 340
14.7 绘制圆形的点 342
14.8 插入图像 342
14.9 制作贴花 342
14.10 用模板缓存绘制填充的凹多边形 343
14.11 确定重叠区域 344
14.12 阴影 345
14.13 消除隐藏线 345
14.14 纹理映射的应用 346
14.15 绘制深度缓存图像 347
14.16 笛里希累（Dirichlet）域 347
14.17 生命游戏（Life Game） 348
14.18 交替使用 glDrawPixels()和 glCopyPixels() 349
第 15 章 高级应用（一）：OpenGL 与 AutoCAD 的图形接口 351
15.1 AutoCAD 简介 351
15.2 DXF 文件分析 352
15.2.1 DXF 文件的基本结构 352
15.2.2 实体段分析 354
15.3 DXF 接口程序的编写 356
15.4 OpenGL 的 DXF 扩展库函数的制作 362
15.5 高级应用实例 363
第 16 章 高级应用（二）：三维随机分形地形的生成 372
16.1 分形的基本原理 372
16.1.1 自相似 373
16.1.2 一维中点变换 373
16.1.3 Diamond-Square 算法 376

16.1.4 蓝天白云的生成 380
16.2 分形地形生成实例 381
16.2.1 程序的运行 381
16.2.2 代码结构分析 382
附录 1 OpenGL 中的操作顺序 405
附 1.1 概述 405
附 1.2 几何操作 406
附 1.2.1 顶点操作 406
附 1.2.2 图元装配 406
附 1.3 像素操作 407
附 1.4 片元操作 407
附 1.5 补充说明 407
附录 2 OpenGL 状态变量 409
附 2.1 查询命令 409
附 2.2 出错处理 410
附 2.3 状态变量的保存和恢复 411
附 2.4 OpenGL 的状态变量 412
附 2.4.1 当前值和相关数据 413
附 2.4.2 变换 413
附 2.4.3 颜色 414
附 2.4.4 光照 414
附 2.4.5 光栅化 416
附 2.4.6 纹理 416
附 2.4.7 像素操作 417
附 2.4.8 帧缓存控制 418
附 2.4.9 像素 419
附 2.4.10 求值程序 420
附 2.4.11 提示 420
附 2.4.12 依赖执行的值 420
附 2.4.13 像素深度 422
附 2.4.14 混合 422
附录 3 OpenGL 实用库 423
附 3.1 纹理映射中的图像操作 423
附 3.2 变换坐标 424
附 3.3 多边形网格化 425

附 3.4　回调机制……425
附 3.5　网格化物体……425
附 3.6　指定回调函数……426
附 3.7　描述网格化多边形……427
附 3.8　渲染球体、圆柱和圆盘……428
附 3.9　NURBS 曲线和曲面……429
附 3.10　错误的描述……430
附录 4　OpenGL 对 X 窗口系统的扩展……431
附 4.1　初始化……431
附 4.2　控制渲染……431
附 4.2.1　管理 OpenGL 渲染内容……431
附 4.2.2　后台渲染……432
附 4.2.3　同步执行……432
附 4.2.4　交换缓存……432
附 4.2.5　使用 X 字体……432
附 4.3　GLX 库函数原型……432
附录 5　OpenGL 辅助库……434
附 5.1　初始化及退出窗口……434
附 5.2　处理窗口和输入事件……435
附 5.3　装载颜色图……435
附 5.4　初始化和绘制三维物体……435
附 5.5　管理后台进程……437
附 5.6　运行程序……437
附录 6　计算法线向量……438
附 6.1　为解析表面确定法线……438
附 6.2　由多边形数据确定法线……440
附录 7　齐次坐标与变换矩阵……442
附 7.1　齐次坐标……442
附 7.2　变换……442
附 7.2.1　顶点变换……442
附 7.2.2　法线变换……443
附 7.2.3　变换矩阵……443
附录 8　OpenGL 编程要点……446
附 8.1　OpenGL 正确编程技巧……446
附 8.2　提高 OpenGL 程序性能……447

附 8.3 GLX 的技巧 448
附录 9 OpenGL 的不变性 449
附录 10 术语表 450
A 450
B 450
C 451
D 452
E 453
F 453
G 454
H 455
I 455
L 456
M 456
N 456
O 457
P 457
R 458
S 458
T 459
V 459
X 460

第 1 章　OpenGL 入门

本章主题

作为图形硬件的软件界面，OpenGL 的主要目的是在帧缓存中渲染二维和三维物体。而这些物体是由定义了几何物体的顶点序列或定义了图像的像素点来描述的。OpenGL 对这些数据通过几步处理，将其变换成帧缓存中最终图像的像素点。

本章系统地介绍了 OpenGL，概览了 OpenGL 如何进行工作。主要由以下几部分组成：

- OpenGL 概览：解释何为 OpenGL，它能实现哪些功能，并介绍了 OpenGL 的执行机制。
- 一个简单的 OpenGL 例程：介绍了一个 OpenGL 的小程序，对其作了简单的讨论。在这一节还给出了一些基本的计算机图形学术语的定义。
- 基本的 OpenGL 操作：给出了关于 OpenGL 怎样处理数据，并在帧缓存中产生相应图像的高层描述。
- OpenGL 的命令语法：阐述了 OpenGL 命令所遵循的约定和符号。
- 作为状态机的 OpenGL：描述了 OpenGL 中状态变量的用法以及用于查询、激活和终止状态的命令。
- OpenGL 相关函数库：简要描述了 OpenGL 相关的库函数集。
- 动画：以通用的术语解释了怎样在屏幕上制作运动的图像——动画。

1.1　OpenGL 概览

OpenGL 是图形硬件的软件接口。OpenGL 中包括大约 120 个不同的命令，程序员可以使用这些命令设定所需的物体和操作，来制作交互式的三维应用程序。

有时，图形的创建和显示并不是在一台计算机上完成，这种情况下 OpenGL 也能有效工作。比如在计算机网络环境下，运行程序和发出 OpenGL 绘图命令的计算机叫作“客户”，接收并执行绘图命令的计算机叫作“服务器”。由于从“客户”到“服务器”发出的命令（称为协议）总是相同的，这种 OpenGL 的网络透明性使得 OpenGL 程序在网络上也能够得以顺利运行。

OpenGL 是作为一种新型的接口来设计的，其与硬件无关的特性使它可以在不同的硬件平台上实现。也是为了实现这一跨平台性能，OpenGL 没有包括执行窗口任务或获取用户输入的命令，程序员必须通过所使用的窗口系统来控制正在使用的特定硬件。作为底层的支持

平台，OpenGL 也不提供描述三维物体模型的高级命令（如绘制汽车、身体的一部分、飞机或者分子模型等）。使用 OpenGL 的程序员必须利用由一系列简单的点、直线和多边形等几何图元的组合来建立期望的模型，也可以在 OpenGL 的顶层建立提供这些高级特性的较为复杂的函数库（如“Open Inventor”，参见“OpenGL 相关函数库”一节）。换一句话说，OpenGL 并不是专长于建立模型，它主要的优势在于渲染。这也就是为什么我们要在本书中专辟一章讨论关于建模平台和渲染平台的接口的问题（参见第 15 章）。

现在读者已经了解了 OpenGL 的几点局限，那么 OpenGL 到底能做些什么呢？为了使读者有一个感性的认识，我们参照 OpenGL 制作的彩图来对 OpenGL 的功能进行一个初步的了解。

彩图 1 显示了整个场景的线框模型图。网格的每一条线对应于图元（一般是多边形）的边，比如：桌子的表面是由三角形组成的，好像被切开的一整块蛋糕。需要注意的一点是，如果物体是实心的，而不是线框组成的，物体将有一部分被遮挡起来看不到。所以尽管实际上山的大部分都被墙壁遮住了，此时仍能够看到窗外整个山的模型。球是由数百个彩色分块组成的，观察者看到的是所有分块的边缘线框，包括球后面各个分块，所以整个球体看起来已经接近于实体。任何复杂的物体，都可以应用这种简单物体的组合的方式来创建。

彩图 2 显示了同一线框场景，但其中加入了深度提示（depth-cue）。读者可以注意到，距离人眼较远的线较为昏暗，这样一来给出了更加接近现实生活的生动的深度提示。

彩图 3 显示了线框场景的另一种形式，在这里进行了锯齿现象的消除。锯齿现象的消除这一技术应用于减少锯齿现象对图像逼真度的破坏（当边缘附近的像素只有一部分属于该图像时，就会产生锯齿现象。这种锯齿现象在近水平或近垂直直线上常表现得尤其明显）。

彩图 4 显示了具有无明暗差别的阴影（又称平影）的场景形式。场景中的物体显示为单色的实心物体。这些物体整个表面显得“无明暗差别”（指其对室内的光照条件并无反应），因此，它们看上去不很平滑。

彩图 5 显示了具有光照及平滑阴影的场景。读者可以注意到，当加在物体上的阴影对室内的光源作出响应时，物体表面看上去非常平滑，场景看起来便更加真实，而且具有三维立体感。

彩图 6 显示了在前面场景中加上阴影和纹理后的效果。阴影特性在 OpenGL 中并没有明确的定义（OpenGL 没有“阴影命令”）。但是程序员可以应用第 14 章所介绍的技术来自己创建。纹理映射的操作使程序员能够把二维纹理应用到三维物体上。这个场景中，桌面就是纹理映射的一个最生动的例子。墙壁、地板、桌面、桌子的上方，都应用了纹理映射。

彩图 7 显示了场景中经过运动模糊化处理的物体。这一技术使物体向前运动时，看起来就好像是被什么抓住了似的，画面上留下了物体运动的模糊轨迹。

彩图 8 显示的是以不同的视点画出的场景。

接下来的两幅彩色图像阐释了应用 OpenGL 能够实现的更为复杂的视觉效果。

彩图 9 显示了大气效应的使用（通常指雾），图中显示出空中存在着微粒。

彩图 10 显示了视野效应，模拟出照相机一种特性——镜头不可能使所有照片场景中的物体处在焦距处。照相机只能对场景中的某一特定点聚焦，比该点近或远的物体，看起来都显得有些模糊。

通过对具体图形的阐释，想必读者对利用 OpenGL 图形系统可以实现的各种效果已经大体上有了一个感性的认识。下面简要描述一下在渲染屏幕图像时，需要 OpenGL 执行的主要图形操作所遵循的次序：

1）以几何图元构造表面，从而对物体进行数学描述（在 OpenGL 中，点、直线、多边形、图形、位图都被看成是图元）。

2）在三维空间布置物体，选择用于观察组合场景的最佳观察点。

3）计算各物体的颜色。在应用程序中可以明确地分配颜色：颜色可以通过指定光照条件来确定，也可以通过在物体上粘贴纹理来获得。

4）将物体的数学描述和相关的颜色信息变换成屏幕像素，这一过程称为光栅化。在这一步，可以进行其他 OpenGL 操作，如：消除物体后的隐藏部分（隐藏部分不画，这样可以提高性能）。另外，在场景进行了光栅化之后、但是还没有绘制到屏幕上时，程序员可以按照自己的意愿对像素数据进行处理。

在附录 1 中，对操作顺序进行了更为详尽的阐述。

1.2　简单的 OpenGL 程序举例

应用 OpenGL 图形系统来完成的事情是十分庞杂的，所以有时一个 OpenGL 程序可能相当复杂。但是对于许多大程序，其基本结构是十分简单的——比如程序的任务就是初始化某些状态来控制 OpenGL 的渲染，并且指定要渲染的物体。所以在程序编制之前，弄清程序结构是十分有益的。

在研究 OpenGL 例程之前，读者需要再熟悉几个术语。

渲染，指计算机根据模型创建图像的过程；模型或物体，是由几何图元——点、直线、多边形等——构成的；图元可以绘制为几种不同模式。对每一种模式可以进行独立的控制（即设定一种模式不会影响其他模式的设置，最终帧缓存的结果是由模式的共同作用来决定的）。通过函数调用的形式来发送命令，可以指定图元，设置模式，或进行其他 OpenGL 操作。

这些几何图元通过指定其顶点加以确定。数据（包括顶点坐标、颜色、法线、纹理坐标和边缘标志）与顶点相关联，每个顶点和其相关的数据都单独的、依次的、以同样的方式进行处理。这一准则有一个例外：为使图元尺寸适合某个特定区域而必须裁剪掉一些顶点时，顶点数据必须进行修改并产生新的顶点。裁剪的类型取决于顶点表示的是什么图元。尽管在命令作用前可能有一个不确定的延迟，总的来说，命令的处理是按照它们发送的顺序进行的。这意味着在后面的命令起作用前，能够完整地画出每个图元。这也意味着状态查询命令所返

回的数据，是与前面发送的所有 OpenGL 命令的执行结果相一致的。

最终渲染的图像由屏幕上绘制的像素组成；像素（pixel，图像元素 picture element 的缩写）是由显示硬件能够输出到屏幕上的最小的可视元素。像素信息（比如像素绘制所采用的颜色）在系统内存中组织为位平面。位平面是一块内存区域，它为屏幕上的每一个像素保存了一位的信息；比如，这一字节信息可能代表像素的颜色红的程度。位平面本身组成了帧缓存，帧缓存中包含了图形显示需要的来控制屏幕所有像素亮度的信息。

OpenGL 提供了相当直接的对二维或三维图形的基本操作控制，包括设定变换矩阵、光照方程系数、锯齿现象的消除方法和像素更新等操作的参数。然而，它并没有提供描述复杂几何体的方法。因此，发送的 OpenGL 命令指定了如何得到某种结果（遵循何种过程），而不是描述这种结果看起来像什么。这就是说，OpenGL 从本质上是程序性的而不是描述性的。正是这种程序性帮助我们了解 OpenGL 怎样工作（如执行操作的次序），从而能够更好地使用它。

现在让我们看一个 OpenGL 例程的基本框架。

例 1-1 是在黑色背景上对一个白色矩形进行渲染，其结果如图 1-1 所示。

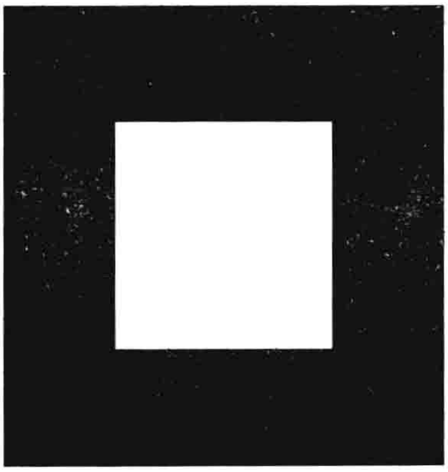


图 1-1 黑色背景下的白色矩形

例 1-1 一个简单的 OpenGL 例程主框架

```
#include <whateverYouNeed.h>
main()
{
   OpenAWindowPlease();
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
   glBegin(GL_POLYGON);
```

```
        glVertex2f(-0.5, -0.5);
        glVertex2f(-0.5, 0.5);
        glVertex2f(0.5, 0.5);
        glVertex2f(0.5, -0.5);
    glEnd();
    glFlush();
    KeepTheWindowOnTheScreenForAWhile();
}
```

主程序第一行是在屏幕上打开一个窗口：函数 OpenAWindowPlease()是窗口系统指定子程序的占位符。接下来的两行 OpenGL 命令将窗口清为黑屏：glClearColor()命令设定窗口清空颜色；而 glClear()实际进行清空操作。清空颜色一旦设置，无论何时调用 glClear()命令，窗口都将被清为该颜色。类似地，glClearColor()命令用于确定以何种颜色绘制物体——在本例中，绘图颜色为白色。从这一条语句开始，直到另一次颜色设置函数调用为止，绘制的所有物体都使用这种颜色。

在这个程序中使用的下一条命令为 glOrtho()，该命令用来指定 OpenGL 用以绘制最终图像的坐标系，同时指定了图像如何映射到屏幕上。接下来的调用都括在 glBegin()和 glEnd()这对命令中间，定义了要绘制的物体——在本例中，要绘制的物体是一个有 4 个顶点的多边形。多边形的“角”由 glVertex2f()命令给出定义。根据参数，即（x,y）坐标组，可以判断出这个多边形是矩形。

最后，glFlush()保证了绘图命令能够得以实际执行，而不是存储在缓冲区中等待另外的 OpenGL 命令。占位程序 ForAWhile()强迫画面保留在屏幕上，而不是马上消失。

完整的程序请见例 1-2。

1.3　基本 OpenGL 操作

图 1-2 给出了 OpenGL 处理数据的高层模块简要框图。图中，函数从左侧进入，通过一系列类似管道的处理过程。一些命令指定要绘制的几何物体，另一些在不同的操作阶段控制对物体的处理。

如图 1-2 中第一个模块所示，OpenGL 并不让每个函数立即通过处理管道，而是把它们累积在一个显示列表中，稍后一次性进行处理。

处理管道的求值阶段，通过输入数据的求值多项式，提供了一种高效的方法来近似绘制几何曲线和曲面。下一阶段进行对每个顶点的操作和图元组合。对顶点进行变换和光照处理，对图元进行裁剪以适合视区大小，这样便为下一阶段的处理做好了准备。

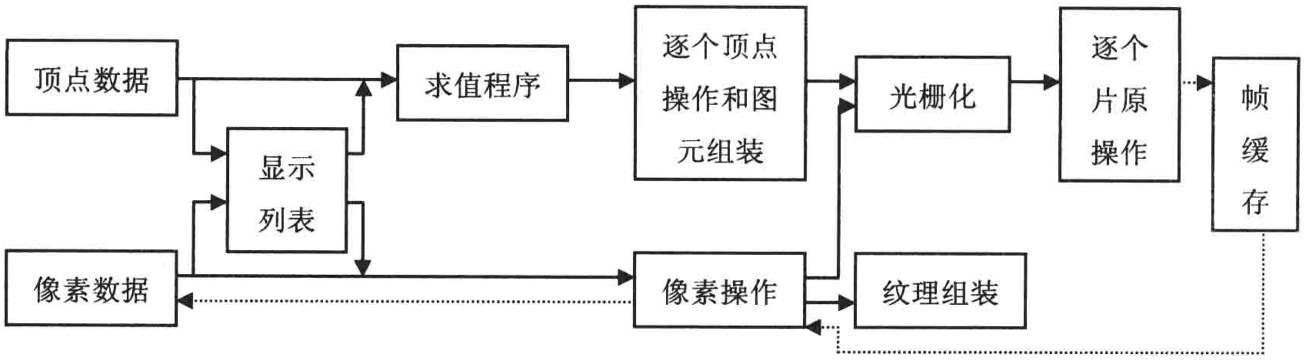


图 1-2　OpenGL 模块框图

在光栅操作中，通过点、线段、多边形的二维描述，产生一系列的帧缓存地址和相关数值。这样产生的每一片原都传入最后的处理阶段——“逐个片原操作”，对数据进行最后的处理，然后将它们作为像素存储在帧缓存中。这些操作包括：基于输入和先前储存的 z 值对帧缓存（z 缓存）进行有条件的更新，混合输入像素颜色与存储颜色、屏蔽等其他对像素值的逻辑操作。

输入数据也可以是像素形式，而不是顶点形式。这类数据（可以描述用于纹理映射中的一幅图像）将跳过上述处理的第一阶段——“求值程序”，作为像素值直接在像素操作阶段进行处理。这一阶段的处理结果，或者存储为纹理内存以用于光栅操作阶段，或者采取同几何数据一样的形式进行光栅化，并将结果片原融合入帧缓存中。

所有 OpenGL 状态的要素，包括纹理内存的内容表，甚至帧缓存，都可以在 OpenGL 管道运行过程中获得。

更详细的内容读者可以进一步参考“库函数详解”中的相关内容。

1.4　OpenGL 的命令语法

从前一节的简单例程中，读者可能已经观察到，所有 OpenGL 的命令都使用前缀“gl”和词首字母大写的单词共同组成命令名（如 glClearColor()）。类似的，OpenGL 定义的常量中，都以 GL_开头并且所有的字母都大写，单词之间以下划线来分隔（如 GL_COLOR_BUFFER_BIT）。

读者可能同时注意到了在某些命令后面，附加有一些看起来不相关的字符（比如，glColor3f()中的后缀 3f）。确实，作为设置当前颜色的命令的定义，该命令名中的 Color 部分就已经足够了。然而，为了使得程序员可以使用不同类型的参数，定义的这类命令不止是一个。详细地说，后缀中的 3 表示给出的参数为 3 个； Color 命令的另一种形式带有 4 个参数。后缀中的 f 表示参数为浮点型数值。某些 OpenGL 命令接受多至 8 种不同的数据类型作为它们的参数。表 1-1 列出了对于 OpenGL 的 ANSI C 实现，用作后缀以指定参数数据类型的字母，以及相应的 OpenGL 数据类型定义。程序员所采用的 OpenGL 特定实现不一定严格

遵循这种方案；比如：OpenGL 利用 C++ 或 Ada 的实现，就未必遵循这种方案。

因此，下面两个命令：

```
glVertex2i(1, 3);
glVertex2f(1.0, 3.0);
```

是等价的，除了第一条命令以 32 位整型数指定顶点坐标，而第二条命令则是以单精度浮点数来指定。

表 1-1　命令后缀参数数据类型

后缀定义	数据类型	相应的 C 语言类型	OpenGL 类型定义
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned long	GLuint, GLenum, Glbitfield

有一些 OpenGL 命令的最后面带有一个字母 v，这表示该命令带有的是一个指向矢量（或数组）值的指针参数，而不是一系列单独的参数。许多命令都有矢量和非矢量两种参数形式，但是一些命令只接受独立参数，而另一些命令则要求至少某些参数要指定为矢量。下面几行命令例示了设置当前颜色命令的矢量与非矢量形式。

```
glColor3f(1.0, 0.0, 0.0);
float color_array[] = {1.0, 0.0, 0.0};
glColor3fv(color_array);
```

在本书的其余部分（实际的源代码例程除外），仅以其基本命名表示 OpenGL 命令，而在末尾包括进来一个“*”号表示命名中的其他部分。比如，glColor*()代表设置当前颜色命令的所有变化形式。如果我们希望对某一特定命令的一种形式做一特殊的说明，则将定义该形式的必要后缀包括进去。比如：glVertex*v()命令代表的是用于指定顶点的命令的所有矢量形式。

最后，OpenGL 定义了常量 Glvoid；如果用 C 语言编写程序，读者可以使用该常量来代替 void。

1.5　作为状态机的 OpenGL

OpenGL 是一个状态机。这就是说，将 OpenGL 设置为各种状态（或各种模式）后，这

些状态将一直有效地保存在 OpenGL 中，直到改变了这些状态的值为止。正如读者已经看到的那样，当前颜色就是一个状态变量。可以将当前颜色设置为白色、红色或其他任意一种颜色，接下来所有的物体都将以该颜色画出，直到程序员改变了当前颜色设置为其他的某种颜色为止。当前颜色仅仅是 OpenGL 保存的多种状态变量之一。其他的状态变量控制另外的状态，如当前视图和投影变换、直线和多边形点划模式、多边形绘画模式、像素包装协议、灯光的位置和特征、被画物体的材料属性等。许多状态变量都指的是可以使用命令 glEnable()或 glDisable()来激活或取消的模式。

每一个状态变量或模式都有一个缺省值，程序员可以在程序中的任意位置对系统进行查询，以获得每一个变量的当前值。一般地，程序员可以使用下列四个命令之一来完成查询操作：gLGetBooleanv()，glGetDoublev(), glGetFloatv()或 glGetIntegerv()。采用哪一个命令取决于程序员期望以何种数据类型给出查询结果。某些状态变量有更为特别确定的查询命令（如 glGetLight*()，glGetError()或 glGetPolygonStipple()）。

另外，程序员可以用 glPushAttrib()和 glPopAttrib()命令将有关属性堆栈的状态变量集的值保存起来，可以随时进行恢复。一般情况下，由于用 glPushAttrib()和 glPopAttrib()更为有效，要尽量使用这些查询命令而不是其他的任意命令。

可供查询的状态变量的完整列表见附录 2。对每一个变量，该附录还列出了返回变量值的 glGet*()命令，其所属属性类，以及变量的缺省值。

1.6 OpenGL 相关函数库

OpenGL 提供了功能强大、操作简单的一套渲染命令，所有的高级绘图必须通过这些命令来完成。因此，为了简化特定的程序任务的编写与执行，程序员可以在 OpenGL 的顶层编写自己的函数库。同样，程序员还可以编写一些子程序，来使 OpenGL 程序更具有针对性，容易在自己的窗口系统中得以运行。事实上，这类可以提供特殊特性的函数库和子程序，有一些已经编制出来了。这些库在下面将进行初步的介绍。

1.6.1 OpenGL 相关函数库简介

OpenGL 实用库（GLU）中的函数通过底层 OpenGL 命令来完成这样一些任务，如为特定的观察方位和投影设置矩阵、完成多边形的网格化，以及渲染表面等等。这个库是作为 OpenGL 实现工具的一部分提供的。在附录 3 及库函数详解中对其进行了详细的描述。其他的一些更为实用的 GLU 子函数将在本书的相关章节进行讨论。GLU 子程序使用前缀 glu 来标志。

OpenGL 对 X 窗口系统的扩展库（GLX）提供了创建 OpenGL 的内容表，并使其与计算机（该计算机使用 X 窗口系统）上的可画窗口相联系的一种手段。GLX 是 OpenGL 的一个附件。在附录 4 和库函数详解中对其进行了更详细的阐述。GLX 库中的一个子程序（用于

交换帧缓存）将在动画一节中进行描述。GLX 库函数使用的前缀为 glX。

OpenGL 辅助库将在附录 5 及库函数详解中进行详细的讨论。辅助库子程序使用的前缀为 aux。

Open Inventor 是面向对象的编程工具包，它基于 OpenGL，提供了用于创建交互式三维图形应用程序的物体和方法。Open Inventor 也是由 SGI 公司（Silicon Graphics Inc.）开发的，用 C++进行编写，提供了预先创建的物体和用于用户交互的嵌入式事件模型，同时提供了用于创建及编辑三维场景的高级应用组件，以及进行物体打印、与其他图形格式交换数据的能力。

1.6.2　OpenGL 辅助库（GLAUX）

尽管 OpenGL 包括渲染命令，但却是独立于任何窗口系统或操作系统而设计出来的。因此，OpenGL 并不包括用来打开窗口以及从键盘或鼠标读取事件的命令。只要编写一个完整的图形程序，就必须打开一个窗口，要不然无法进行任何操作。许多程序都要求一些用户输入，或者需要操作系统或窗口系统提供其他的一些服务。OpenGL 辅助库便可以用来简化诸如打开窗口、检测输入等操作。

由于 OpenGL 的绘图命令仅限于生成简单的几何图元（点、直线和多边形），因此，辅助库包括了一些用于创建更复杂的三维物体，如圆球、圆环及茶壶等物体的子程序。这样，不但使程序的编制变得轻松，而且还将增加不少乐趣。OpenGL 发展至今，辅助库已经作为其一个部分在几乎所有的编译平台中都集成了进来，读者已不必担心找不到 GLAUX 而在复杂物体绘制上大伤脑筋了。

辅助库的设计者们编制的这些函数十分简单易用，本书中的例程都是建立在辅助库的基础之上的。虽然以它为基础实现大的应用程序仍然是相当困难的，但是从学习的角度来看，这将是一个良好的开端。本节的剩余部分简要地描述了这个辅助库，以便于读者跟着本书后面的例程开始学习。关于这些程序的详细说明请参见附录 5 与库函数详解。

1. 窗口管理

为了初始化并打开一个窗口，需要调用三个函数完成必要的任务。

- auxInitWindow()命令，用来在屏幕上打开一窗口，窗口名称在函数参数中给出。Esc 键将被激活，以随时终止程序。同时将窗口背景颜色设置为缺省黑色。
- auxInitPosition()命令，通知 auxInitWindow()命令应该把窗口放置在屏幕上什么位置。
- auxInitDisplayMode()命令，通知 auxInitWindow()命令是创建 RGBA 模式窗口还是创建颜色索引模式窗口。程序员还可以指定一个单缓存或双缓存窗口。如果在颜色索引模式下工作，有时要将某种颜色加载到颜色表中；这一任务可以使用 auxSetOneColor() 命令来完成。也可以使用该程序来指示出程序员希望该窗口具有的相关深度、模板和/或累积缓存。

2. 处理输入事件

程序员可以利用下列函数来注册回调函数，这一回调函数将在指定事件发生时加以激活。

- auxReshapeFunc()命令，表示在窗口尺寸改变、移动或暴露时应该执行的动作。
- auxKeyFunc()命令和 auxMouseFunc()命令。在键盘或鼠标被压下或被释放时，这两个命令将键盘键值或鼠标按键与调用的程序联结起来。

3. 绘制三维物体

OpenGL 辅助库包括下列一些绘制三维物体的命令。

- 球体（sphere）
- 立方体（cube）
- 圆环面（torus）
- 圆柱体（cylinder）
- 圆锥体（cone）
- 八面体（octahedron）
- 十二面体（dodecahedron）
- 二十面体（icosahedron）
- 茶壶（teapot）

程序员可以以线框或是定义了表面法线的有阴影效果的实体两种不同形式来绘制物体，例如，下面是绘制圆球和圆环的函数。

```
void auxWireSphere(GLdouble radius);
void auxSolidSphere(GLdouble radius);
void auxWireTorus(GLdouble innerRadius, GLdouble outerRadius);
void auxSolidTorus(GLdouble innerRadius, GLdouble outerRadius);
```

绘制的所有模型都以原点为中心。当以单位比例因子进行绘制时，这些模型都内接于坐标从-1 到 1 的正方体。通过对函数参数的调整可以对物体进行缩放。

4. 管理后台进程

程序员可以用函数 auxIdleFunc()指定一个函数，在没有其他事件未决（pending）的时候——比如当事件循环就要空闲时，便执行这个函数。该函数以一个指针指向后台函数作为其惟一的参数。传递参数 0 时将取消函数的执行。

5. 运行程序

在主程序 main()中，调用 auxMainLoop()命令，并把重画场景物体的函数名作为参数传递给该命令。例程 1-2 例示了如何使用辅助库创建图 1-1 所示的简单程序，此处是在白色的屏幕上绘制一个蓝色的矩形。

例 1-2 利用辅助库编写一个简单的 OpenGL 程序：simple.c

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glaux.h>
void main()
{
```

```
    // OpenAWindowPlease();
   auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
   auxInitPosition (0, 0, 500, 500);
   auxInitWindow ("编程指南例程 1-1");

    glClearColor(1.0, 1.0, 1.0, 0.0); // 指定清空颜色
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f(0.0, 0.0, 1.0);     // 指定绘制物体颜色
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);        // 正交投影模式
    glBegin(GL_POLYGON);
        glVertex2f(-0.5, -0.5);  // 此处可以用 glColor3f 设置
        glVertex2f(-0.5, 0.5);   // 每个顶点的颜色生成一个
        glVertex2f(0.5, 0.5);    // 五颜六色的多边形
        glVertex2f(0.5, -0.5);

    glEnd();
    glFlush();
   // KeepTheWindowOn-
   // TheScreenForAWhile();
    _sleep(2000);
}
```

运行结果如图 1-1 所示。

1.7　动画

在图形计算机上，运动画面的绘制可能是最令人感兴趣的事情。无论对于一位试图查看所设计的机械零件所有侧面的工程师，还是利用模拟器学习飞机驾驶的飞行员，或者仅仅是一位计算机游戏迷，动画都是人们头脑中的计算机图形学的一个重要的组成部分。

在电影院里，通过以一定的速度（每秒 24 帧）放映一系列的画面，同时将其以每秒 24 帧的速度投影到屏幕上，就可以得到运动的画面。当每一帧画面移到镜头后面时，快门打开，这一帧画面就显示出来。胶片向下一帧移动时，快门立即关闭，到了画面处快门再次打开将该画面显示出来，如此反复。尽管观众每秒钟看到的是 24 帧不同的画面，但是大脑中的视觉暂留将其混合成一段连贯的动画。早期的查理•卓别林的电影中，以每秒钟 16 帧进行放

映，这使画面看起来有很明显的闪烁跳动。多数的现代放映机，常常以每秒 48 帧的速度将每帧画面显示两次，这样一来就大大减少了闪烁。图形计算机的屏幕刷新（重新绘制画面）的频率一般大约是每秒钟 60 到 76 次，有的甚至以每秒 120 赫兹的频率进行刷新。很明显，60 赫兹比 30 赫兹更连贯，而 120 赫兹比 60 赫兹的效果更佳。然而，超过 120 赫兹的刷新频率已经超过了衰减返回点——人的眼睛的性能仅此而已。

制作运动画面投影的关键思想是：在显示每一帧画面时，该画面应是完整的。如果用类似下面的程序制作有百万帧画面的计算机动画。

```
open_window();
for (i = 0; i < 1000000; i++)
   clear_the_window();
   draw_frame(i);
   wait_until_a_24th_of_a_second_is_over();
}
```

由于系统清屏和绘制关键帧画面都需要时间，而在这里未加考虑，所以这个程序画出的将是有闪动的画面，闪动的程度取决于清屏和绘图所花的时间与 1/24 秒的接近程度。假定作图所花的时间与 1/24 秒几乎接近，那么最先绘制的各物体在 1/24 秒内是可见的，并且在屏幕上显示为一幅实体图像；当程序运行到下一帧时，先前所画的各个物体立即被清除了，因此这样显现的图像是非常可怕的——在这个 1/24 秒中的大部分时间，眼睛观察到的是清除后的背景，后面需要画出的那些物体正在缓存内进行绘制。问题在于该程序没有把绘制的关键帧画面显示完整，而是边绘制边进行观察。

解决上述问题的一个简单的途径是提供双缓存——提供两个完整的颜色缓存的硬件或软件。当一个缓存用来作图时，另一个用来显示。当画完一帧时，交换两个缓存。这样，刚才用来显示的缓存现在用来作图。这就像一个只以两帧进行循环的电影放映机，当一帧画面被投影到屏幕上时，有一位画家要迅速地擦掉这幅画面并重画另一帧。只要画家的动作足够快，观众就注意不到这套设备与只是将事先已经画好的一帧一帧画面简单地进行播放有什么差别。这在实际中当然是不可能做到的，然而计算机可以。应用双缓存，每一帧画面只在绘制完成之后才显示出来，因此观察者不会看到只绘制了一部分的画面。

将前面的程序稍作修改，便可以连贯性地显示动画图形。程序如下：

```
open_window_in_double_buffer_mode();
for (i = 0; i < 1000000; i++)
   clear_the_window();
   draw_frame(i);
   swap_the_buffers();
}
```

除了简单地在视图缓存和绘图缓存之间进行切换以外，为了保证前面的缓存得以完整显

示，程序中的 swap_the_buffers()函数将一直等到当前屏幕更新期结束才进行循环的下一步。该函数还允许从头开始完整地显示一个新的缓存。假设一个系统每秒更新 60 次，这就意味着以最快的速度，在一秒内最多可以显示 60 帧画面，如果两个帧缓存都可以在 1/60 秒之内清除和画出，那么动画就能够以该速度平滑地运行。

在这样的系统中经常发生这种情况：画面太复杂，在 1/60 秒内无法画出，这样，每一帧的显示将超时。另外，如果绘制一帧要 1/45 秒，而每秒只放映 30 帧，这样每放映一帧画面就出现了 1/30-1/45 =1/90 秒的空闲时间。尽管浪费 1/90 秒是很短的时间，但是每 1/30 秒就浪费了 1/90 秒，这意味着实际上有 1/3 的时间被浪费掉了。

另外，视频的更新速度是不变的，这就可能产生一些意想不到的运行结果。比如，以 1/60 秒的显示器刷新速度以及恒定的帧速，程序员能够以 60，30，20，15，12 帧/秒等（60/1，60/2，60/3，60/4，60/5……）速度运行程序。这意味着，如果程序员正在编写一个应用程序，并逐渐加入某些特性（比如一个飞行模拟器，正在增加地面场景），最开始所增加的特性对整体性能都没有什么影响——仍旧可以获得 60 帧/秒的速度。但是突然，在又加入一个新的特性后，由于系统已经不能在 1/60 秒内绘制完整画面，这样就错过了第一次可能进行缓存交换的时间，只好等到下一次进行，应用程序的性能将降低一半——每秒显示 30 帧。当每一帧的绘制时间超过 1/30 秒时，又会发生类似的事情——显示速度从每秒 30 帧降到了 20 帧，性能降低了 33%。

另一个问题就是如果根据场景的复杂程度进行的绘制时间与任何特殊的时间间隔接近时（本例中 1/60 秒，2/60 秒，3/60 秒等等），由于变化的随机性，某些帧将稍有超时，而某些帧又用不满这个时间，因此帧速度是不规则的，这样就会出现视觉上的扰动。在这种情况下，如果不能通过简化场景来保持足够快的帧速度，那么最好的办法就是有意为每一帧增加一个觉察不到的极小的延迟，以便保持一个稳定的稍慢的帧速度。如果画面之间的复杂度有剧烈的变化，那么就可能需要一种更复杂的算法了。

在实际应用中，真正的动画程序的结构与这里的描述极为类似。通常，对每一帧画面整个缓存都要从轮廓图开始重新绘制，这往往比仅计算哪部分需要重画更容易，而且在许多情况下，这种办法都是更加实用的。比如对于三维飞行模拟器，飞机方位的一个微小的变化会引起窗外几乎所有事物的位置都发生变化，如果计算变化的部分来确定重画的范围，计算量显然是十分大的。

在大多数动画中，屏幕上物体的重画是简单地采用不同的变换——随着观察者视点的移动，或是汽车沿公路一点一点的移动，或是物体慢慢地旋转。如果结构要做重大的修改，每一帧画面都需要进行重新计算，那么帧速度常常会变慢。记住，在 swap_the_buffers()命令执行之后的空闲时间常可以用于进行这样的计算。

OpenGL 中并没有 swap_the_buffers()命令，因为这一特性并不一定在所有硬件上都有效。而且，在任何情况下，这一特性都很大程度地取决于窗口系统。然而，GLX 提供了在运行 X 窗口系统的计算机上使用的命令：

```
void glXSwapBuffers(Display *dpy, Window window);
```

例 1-3 例示出在一个绘制持续旋转的正方形的例子中，**glXSwapBuffers()**命令的使用，其结果如图 1-3 所示。

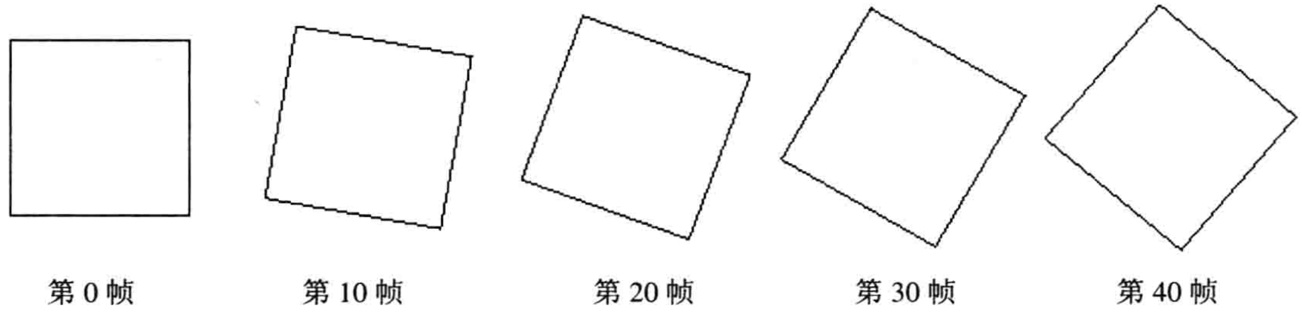


图 1-3 具有双缓存的旋转正方形

例 1-3 具有双缓存的程序：double.c

```
#include <windows.h>

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

static GLfloat spin = 0.0,s = 0.1;
static GLint flag=0;

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    if(flag==0)  //控制绘制矩形在动画转动中由小变大，再由大变小
        s+=0.005;
    else
        s-=0.005;
    if(s>1.0)
        flag=1;
    if(s<0.1)
        flag=0;

    glPushMatrix();
```

```
    glScalef(s,s,s);// 比例变换

    glRotatef(spin, 0.0, 0.0, 1.0);
    glRectf(-25.0, -25.0, 25.0, 25.0);
    glPopMatrix();
    glFlush();
    auxSwapBuffers();//  前后缓存交换
    _sleep(100);
}

void CALLBACK spinDisplay(void)
{
    spin = spin + 2.0;
    if (spin > 360.0)
        spin = spin - 360.0;
    display();
}

void CALLBACK startIdleFunc(AUX_EVENTREC *event)
{
    auxIdleFunc(spinDisplay);
}

void CALLBACK stopIdleFunc(AUX_EVENTREC *event)
{
    auxIdleFunc(0);
}

void myinit(void)
{
    glClearColor(1.0, 1.0, 1.0, 1.0);//白色背景
    glColor3f(0.0, 0.4, 1.0);// 海蓝色物体
    glShadeModel(GL_FLAT);
}
```

```
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho (-50.0, 50.0,
                 -50.0*(GLfloat)h/(GLfloat)w,
                  50.0*(GLfloat)h/(GLfloat)w, -1.0, 1.0);
    else
        glOrtho (-50.0*(GLfloat)w/(GLfloat)h,
                  50.0*(GLfloat)w/(GLfloat)h, -50.0, 50.0,
                 -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
}

int main(int argc, char** argv)
{
     auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例程 1-3");

     auxReshapeFunc(myReshape);
     auxIdleFunc(spinDisplay);
     myinit();
     auxMainLoop(display);
     return 0;
}
```

第 2 章　Windows 95 及 Windows NT 环境下开发 OpenGL 应用程序的基本过程

本章主题

本章着眼于用户直接编制 OpenGL 应用程序的工具，详细介绍了在 Windows 95 及 Windows NT 环境下开发应用程序的基本过程。在这两种环境下，用户可以简单快捷地开发出 Win32 控制台风格和 Windows MFC 风格两种应用程序。本章包括的主要内容如下：

- OpenGL 程序编制所需依赖的开发环境。
- OpenGL 应用程序开发的基本过程。

2.1　开发环境

经过多年实践考验，OpenGL 已经发展为成熟的图形开发平台。而最近以来，OpenGL 在诸多微机编译平台中的拓展，更显示了它强大的生命力。在这一章里，我们将对其微机应用开发环境进行介绍，同时向用户推荐本书程序编制所借助的开发环境。

2.1.1　系统硬件配置要求

虽然 OpenGL 本身是独立于硬件系统、窗口系统及操作系统的，但考虑到我国 PC 及兼容机的广泛应用，在这里我们着重介绍在微机平台上的 OpenGL 开发。

在 PC 及兼容机上开发 OpenGL 应用所需的硬件最低配置如下：

- CPU：Pentium 或 Pentium Pro，主频≥90MHz。
- 内存：≥16MB。
- 硬盘：≥512MB。
- 图形卡：可选。
- 显示卡：无特殊要求。

以上配置是最低配置。配置越高，OpenGL 应用程序的运行及显示效果越佳。

2.1.2　开发 OpenGL 应用程序的软件要求

目前，Microsoft 公司在其 Windows 95 或更高版本的操作系统和 Visual 系列高级语言开发环境中捆绑了 OpenGL 的开发工具。因此，我们建议在下面的环境中开发 OpenGL 应用。

- 操作系统：Windows 95、98，或 Windows NT 3.5 或更高版本。
- 开发平台：Visual C++ 2.0 以上版本，Power Station 4.0（Fortran）。
- OpenGL 库：opengl32.lib，glu32.lib，glaux.lib。在后面我们针对 VC++平台向读者例示如何将几个库加入编译链接环境中。

本章后续部分均采用 Windows 95/Windows NT 4.0 操作系统，和 Visual C++ 6.0 开发平台。

2.2 OpenGL 应用程序开发基本过程

2.2.1 源代码的编写

可以用任何文本编辑器编写 C 或 C++程序源代码。

2.2.2 Win32 控制台风格应用程序的开发

Win32 控制台风格的应用程序，编制简单，编译和运行方便迅速，应用十分广泛。C/C++语言是程序员学习 OpenGL 的十分合适的工具。Win32 控制台风格的应用程序在 Visual C++ 6.0 中的开发过程如下：

1）从开始菜单的程序组中进入 Visual C++ 6.0 开发环境，启动 Visual C++，如图 2-1 所示。

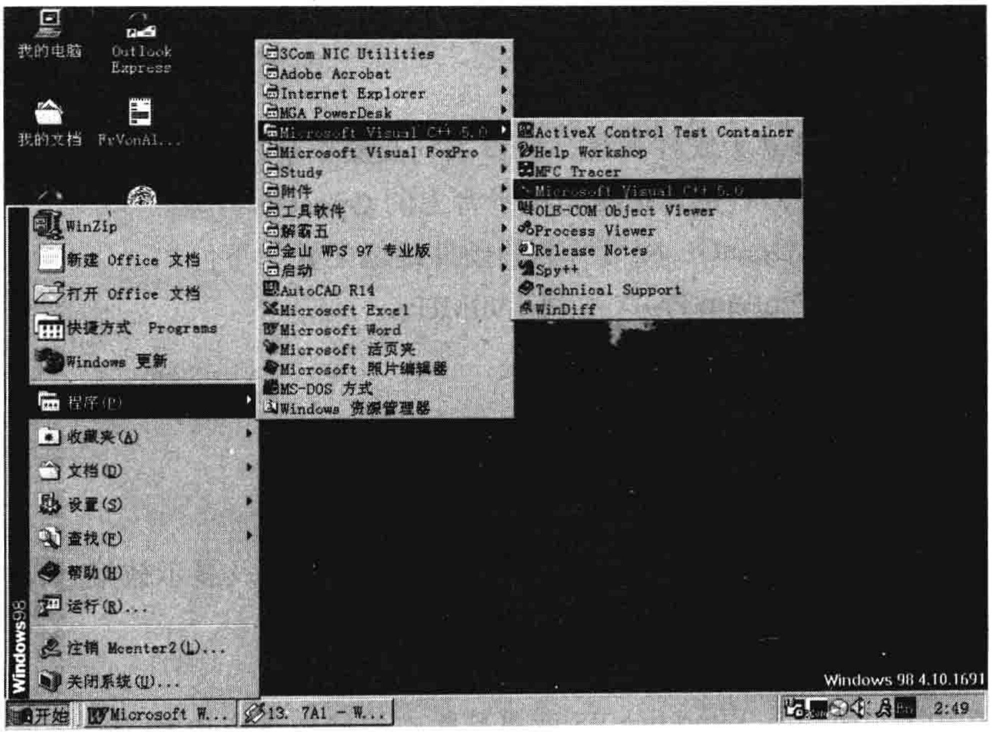


图 2-1 启动 VC++

2）选择 New 项，弹出 New 对话框，如图 2-2 所示。

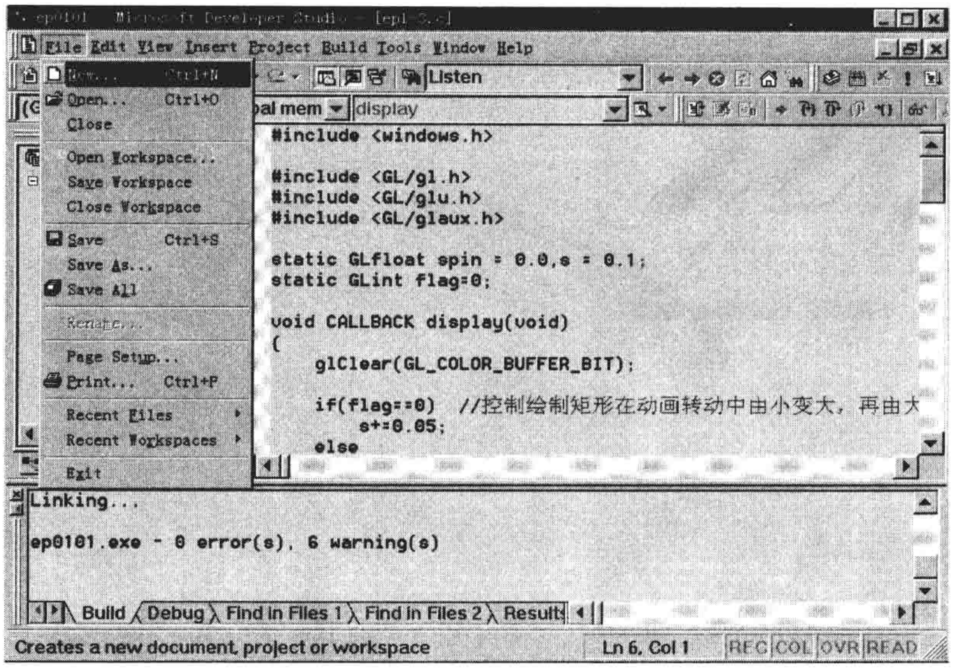


图 2-2　建立新的工程文件

3）在 Projects 标签中，选择 Win32 Console Application 项，在 Project Name 域中键入工程名，在 Location 域中键入工程位置，选中 Create new workspace 项，在 Platforms 域中选中 Win32 项。

4）单击 OK，则新的工程文件创建完成。上述过程如图 2-3 所示。

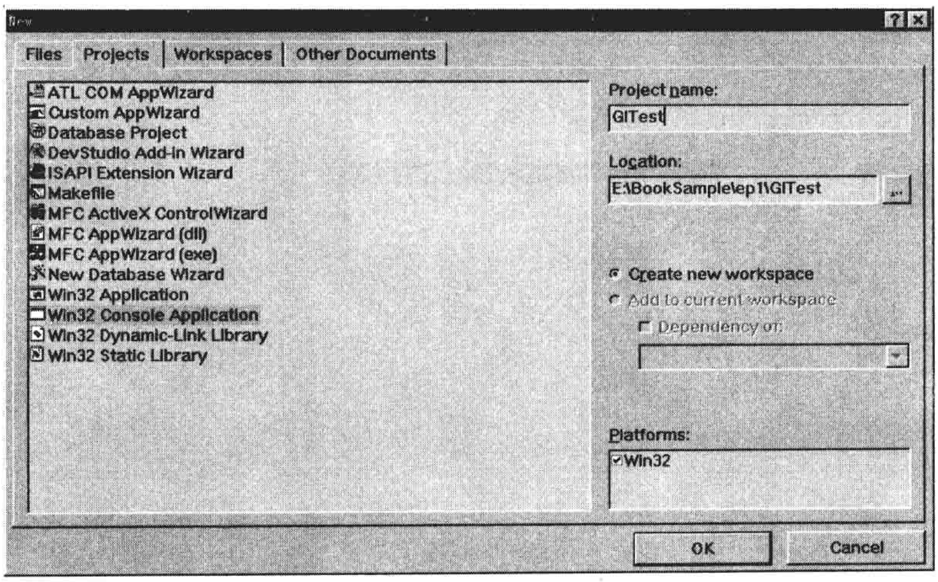


图 2-3　控制台风格的创建工程对话框相应选项

5）向工程中加入源文件。在 Workspace 窗口中，激活 FileView 标签，这时窗口中出现“oglApp files”字样，将光标置于该字样上面，单击鼠标右键，弹出菜单，选中 Add Files to

Project ...（如图 2-4 所示）；或者从 Project 菜单中选中 Add to Project。弹出 Insert Files to Project 对话框。在对话框的文件列表中选择需要的 C 或 C++源文件及相关文件，单击 OK。

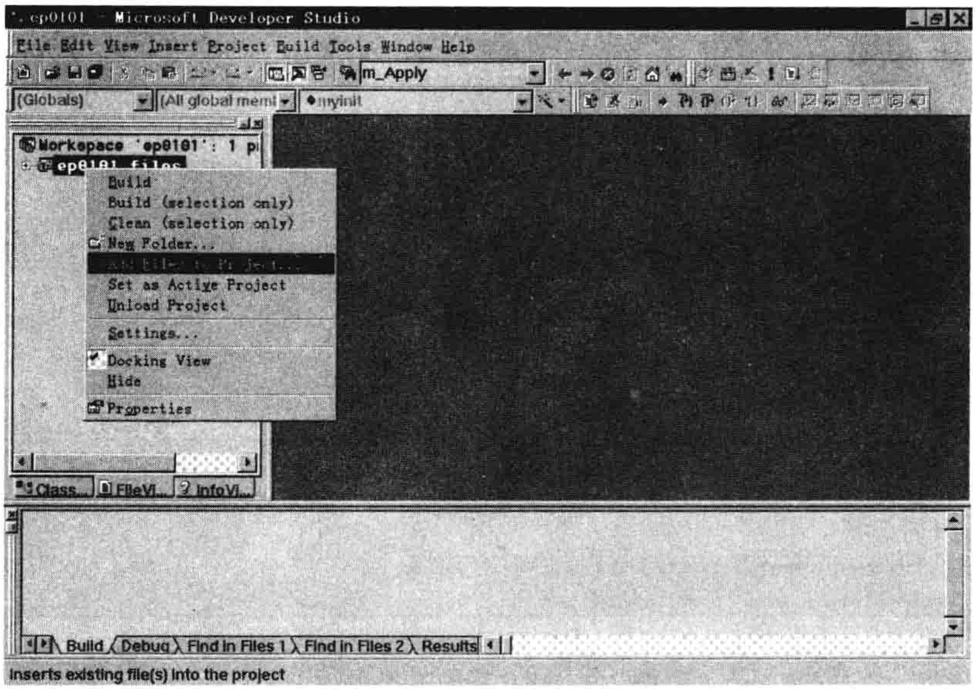


图 2-4　向工程中添加源代码文件

这样，在 FileView 窗口的“oglApp files”文件树下出现添加的文件。

6）设置工程的编译连接选项。先进入 Project 菜单，选中 Settings...，进入工程环境设置，如图 2-5 所示。

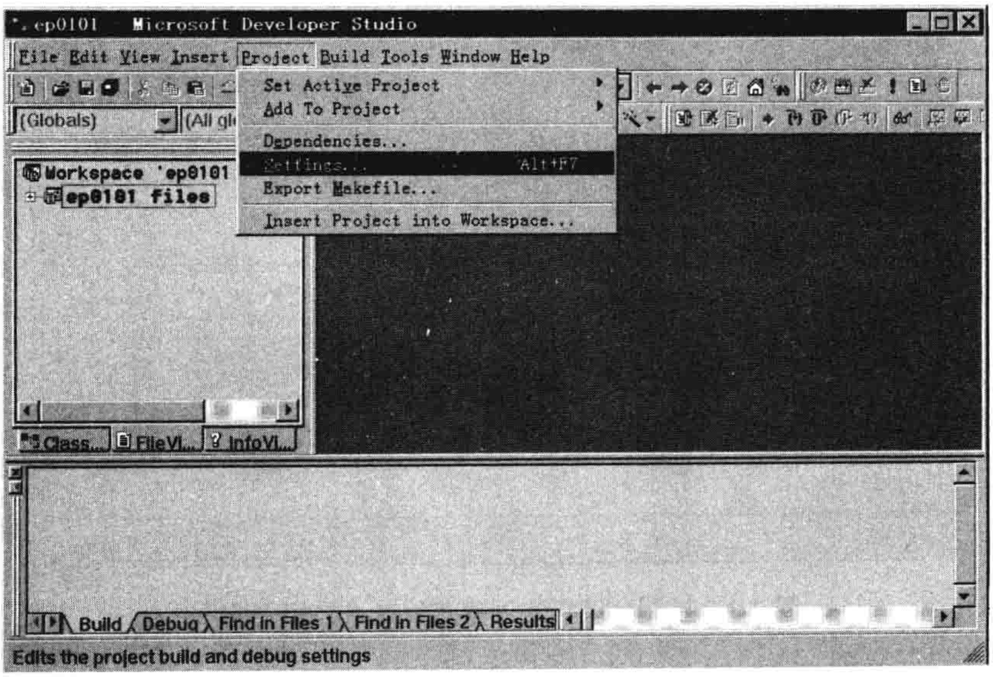


图 2-5　进入编译环境设置

此时弹出 Project settings 对话框。在对话框中，激活 Link 标签，在该标签的 Object/library modules 域中键入 opengl32.lib，glu32.lib，glaux.lib 这三个 OpenGL 的链接库。如图 2-6 所示。

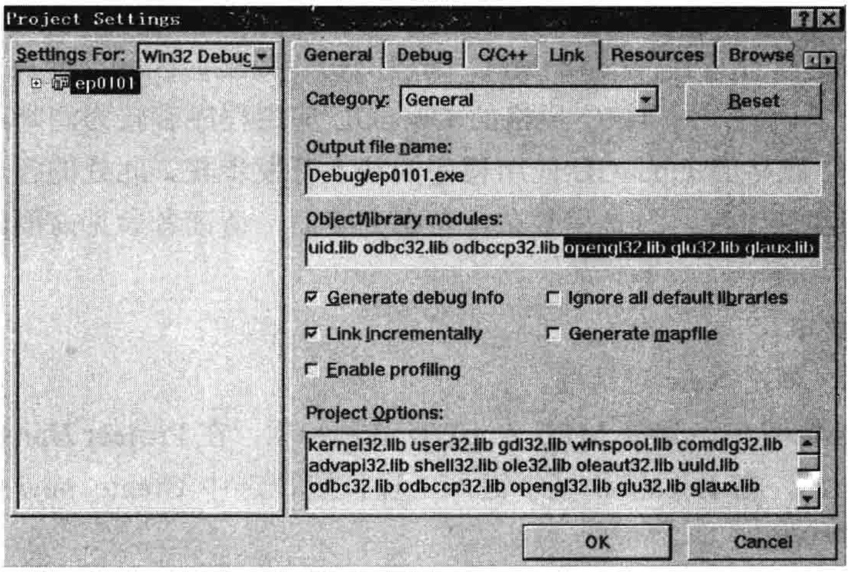


图 2-6　加入链接库

点击 OK 按钮，此工程文件便可以在 OpenGL 要求的环境下得以顺利编译运行了。

7）编译和连接。在 Build 菜单中，选中 Compile 项编译源文件，选中 Build 执行连接，生成可执行文件。也可以直接选中 Rebuild All 项生成可执行文件。

8）运行程序。可以在 Build 菜单下，选择 Execute 执行。也可以直接在控制台运行可执行文件。图 2-7 给出了一个经典的 Win32 控制台风格的 OpenGL 程序。

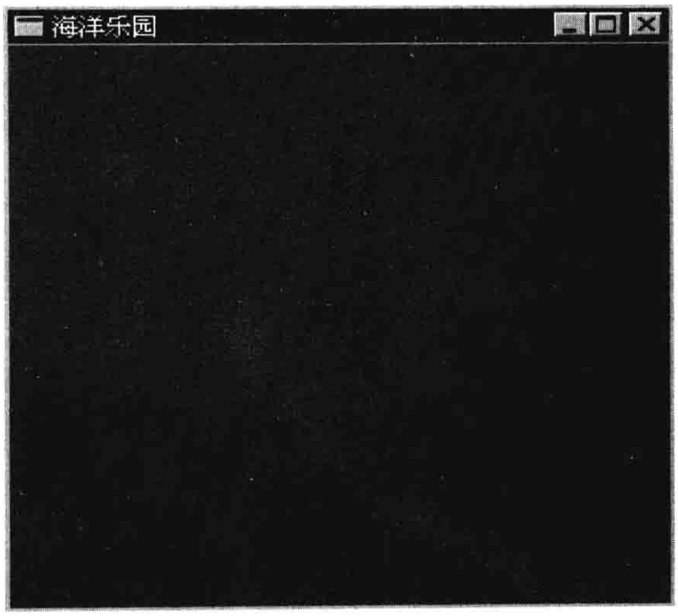


图 2-7　Win32 控制台风格的 OpenGL 应用程序

为方便学习和讲述，本书的所有应用程序都是 Win32 控制台风格的程序。本书中的所有例程都是以这种方式编译的。

2.2.3 Win32 MFC 风格应用程序的开发

这里，我们假定读者具有 Win32 MFC 编程的初步经验，因此不涉及 MFC 风格程序的详细创建过程。为了让用户对 MFC 风格的 OpenGL 应用程序有较为清晰的概念，我们以一个例子来说明 MFC 风格的 OpenGL 应用程序的基本开发步骤。此处的各个步骤类似于前面控制台风格的工程构造过程，只是稍复杂。考虑到版面，请读者参见前面的例图。

1. *创建 MFC 界面*

1）打开 File 菜单。

2）选 New 项，弹出 New 对话框。

3）在 Projects 标签中，选择 MFC AppWizad(exe)项，在 Project Name 域中键入工程文件名，如“oglApp1”，在 Location 域中键入工程位置，选中 Create new workspace 项，在 Platforms 域中选中 Win32 项。

4）单击 OK，弹出 MFC AppWizard—step1 对话框。在对话框中选择 Single document 项，单击 Next。

5）待弹出对话框 MFC AppWizard—step2 of 6 后，单击 Next；一直到对话框 MFC AppWizard —step6 of 6，单击 Finish。这时出现 New Project Information 对话框。在对话框中单击 OK，新工程创建完毕。

2. *在程序中添加 OpenGL 应用代码*

如果我们在这时浏览 Workspace 窗口，会发现 FileView 标签下出现了一组工程文件。工程名为 oglApp1。在“oglApp1 files”处打开文件树，在“Source Files”下面有这几个文件:MainFrm.cpp，StdAfx.cpp，oglApp1.cpp，oglApp1.rc，oglApp1Doc.cpp，oglApp1View.cpp。在“Header Files”中有这几个文件：MainFrm.h，Resource.h，StdAfx.h，oglApp1.h，oglApp1Doc.h，oglApp1View.h。

对于 OpenGL 应用程序来说，上述的文件中最重要的是视图文件 oglApp1View.cpp 及 oglApp1View.h，因为 OpenGL 场景图像最终要向屏幕输出。下面我们要讲述如何向这两个文件中加入 OpenGL 绘图必需的代码。

打开文件 oglApp1View.h。我们可以看到，这个文件用于定义视图类的派生。在本例中，该类名为 COglApp1View。目前，该类中只有视图及文档的基本成员函数。我们在该类中加入下面几个关键的公有成员函数：

```
public:
        void Init();
        void CreateRGBPalette(void);
    BOOL bSetupPixelFormat(void);
```

```
unsigned char ComponentFromIndex(int i, UINT nbits, UINT shift);
void DrawScene(void);
```

在这之后，我们要在文件 oglApp1View.cpp 中对各函数进行定义。

下面，我们简要描述各函数的基本功能。

1）Init()。完成对 OpenGL 的初始化，如视图模式、变换模式等的设置。这些初始设置是通过 glMatrixMode()，glLoadIdentity()，gluPerspective()，glMatrixMode()等函数调用来实现的。特别需要注意的是，在初始化阶段，必须调用 Windows 专用 OpenGL 库函数来建立 Windows 视图与 OpenGL 窗口的联系。这些函数以 wgl 为前缀。在 Init()函数中，这种调用是用下面的两个函数完成的：wglCreateContext()，wglMakeCurrent()。这两个函数将 Windows 与 OpenGL 揉和在一起。

2）CreateRGBPalette(void)。用于设置绘图调色板。

3）bSetupPixelFormat()。为指定的设备内容设置像素格式，关键调用为 SetPixelFormat()。

4）ComponentFromIndex()。该函数用于从颜色索引获取该索引对应的 RGB 颜色分量。

5）DrawScene()。该函数是图形应用程序的主要模块，主要用于绘制各种几何物体。OpenGL 的主要绘图函数一般出现在该模块中。DrawScene()函数的结构与 Win32 控制台程序的主体绘图模块的结构基本一致。因此，对于刚开始学习开发 MFC 风格的应用程序的用户来说，在这一部分进行修改练习完全可以得到基本技巧的训练。

在完成上述成员函数的定义后，另一个问题是如何调用这些函数。在视图文件中有几个实现定义的成员函数，在这些函数中，通常用户要用到的多为窗口创建函数 OnCreate()、绘图函数 OnDraw()及定时函数 OnTimer()。函数 Init()由 OnCreate()函数调用，而 DrawScene()函数由 OnDraw()函数调用。如果程序需要由定时器控制场景的变化，如可视化仿真等应用，那么 DrawScene()函数还要由 OnTimer()函数进行调用。

在中国水利水电出版社网站上有本例的源代码。该程序的运行结果如图 2-8 所示。

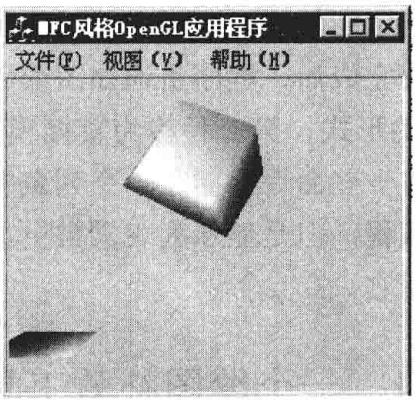


图 2-8　MFC 风格的 OpenGL 应用程序

第 3 章　实体建模基础

本章主题

OpenGL 的操作可以归纳为三种最基本的绘图操作：清空窗口，绘制几何物体及绘制一个光栅物体。光栅物体，诸如二维的图像、位图和字符字体等等，将在第 9 章中讨论。在本章中，读者将学习如何清屏和绘制几何物体（包括点、直线和平面多边形）。

可以利用 OpenGL 绘制的物体是复杂多样的，但这些复杂图形都是由基本图形单位构成的，而且这些图形单位数目并不多。在后面读者会发现，所有曲线和曲面都是由大量的小的直线和平面多边形来近似实现的，分块的大小直接影响着所近似的曲线的光滑程度。在本章的后面，将讲解如何通过大量的小的几何图元构造曲线和曲面。

本章分为以下几个主要部分：

- 绘图补救工具箱，解释了如何清空窗口及强迫绘画结束。同时本节给出了控制几何物体颜色和消除隐藏表面的基本知识。
- 定义点、线和多边形，介绍了基本几何绘图单位及如何进行绘制定义。
- 显示点、线和多边形，解释了程序员如何具体控制图元的绘制，如点的直径大小，直线为实还是虚，多边形是轮廓形式还是填充形式等。
- 法线向量，讨论了如何为几何物体指定法线向量，并简要讨论了这些向量的用途。
- 创建曲面的多边形模型的一些提示，探讨了在利用多边形近似地构造曲面的过程中涉及的问题和技术。

在阅读本章之前，需要提醒读者注意一点：在 OpenGL 中，如果未指定其他物体，那么每发出一个绘图命令，只绘制指定的物体。这样做看起来明显易懂，但一些系统中更适合先将要画的物体列出来，当此过程完成后，程序员通知图形硬件画出列表中的所有项目。

上述的两种组织绘图命令的形式，第一种称为直接模式，是 OpenGL 的缺省形式。除了使用直接模式之外，还可以将一些命令存储在一个列表中（称为显示列表）以待以后绘图使用。直接模式图形显然易于编程，但是显示列表常常更加快捷有效。第 5 章将讲述如何使用显示列表以及为什么需要使用它。

3.1　一个绘图补救工具包

在绘图准备过程中需要清空窗口、设置所画物体的颜色以及强迫完成绘画，尽管这些主题好像与几何物体没有什么直接关系，但是任何绘制几何物体的程序都要涉及这几项内容。

这一节同时介绍了消除隐藏表面的概念，它将使几何体的绘制更加简单。

3.1.1　清除窗口

在计算机屏幕上绘图不同于在纸上绘图，因为纸最初是白色的，绘制者需要做的只是画出线条。在计算机上，载有图画的内存常常被最后画出的那幅图占据着，所以在开始画一个新的场景之前，常常需要以某种背景颜色将其清空。采用何种背景颜色取决于具体的用户应用程序：对于一个文字处理器，希望写入文本之前将背景清为白色（纸的颜色）；若绘制宇宙飞船上看到的场景，在画恒星、行星和其他飞船之前则必须将背景清为宇宙空间的黑色。有些时候是不需要清空屏幕的，例如：一个房间内部的场景，在画出墙壁时，整个图形窗口都将为墙壁颜色所覆盖。

读者可能会有疑问：为什么不画一个足够大的、颜色适当的矩形来覆盖整个窗口，而总是要去清空呢？原因有下面几点：

1）一个特定的清空窗口的命令比通常意义的绘图命令的效率要高得多。

2）在第 4 章中可以得知，OpenGL 允许随意设置坐标系、视图位置和视图方向，所以想要指定一个大小和位置合适的清屏矩形是十分困难的。

3）在 OpenGL 中可以使用隐藏表面消除技术来消除由于更靠近视点的其他物体的遮挡而不必画出的部分物体，如果背景采用清屏矩形的话，就必须保证此矩形位于所画的一切物体之后，在坐标系和视点任意设置的情况下，这是难以做到的。

4）在许多计算机上，图形硬件除了保存要显示的像素颜色的缓存以外，还包括其他的缓存。这些缓存也必须随时加以清除，所以用一条命令可以十分方便地将所有缓存全部清除（所有可能用到的缓存将在第 11 章帧缓存中讨论）。

下面的几行程序代码将把窗口清为黑色。

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

第一行将清空颜色设置为黑色，第二条命令将整个窗口清为当前的清空颜色。glClear()命令中的单参数指出哪一个颜色缓存将被清除。在这个例子中，只有屏幕上的显示缓存被清空。通常，在应用程序中较前的位置设置一次清空颜色，然后随时根据需要清空缓存。OpenGL 以一个全局变量的形式来追踪记载当前清空颜色，而不要求程序员在每次清空缓存时就指定一次颜色。

第 6 章和第 11 章讨论了其他缓存的应用。现在，我们仅需要知道清空缓存是比较简单的。例如，同时清空颜色缓存和深度缓存，只要用到如下的几个命令：

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClearDepth(0.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

在这个例子中，glClearColor()的调用和前面的相同，glClearDepth()命令给定一数值，深

度缓存中的每一个像素都将被设置为这个值，glClear()命令的参数由需要清除的所有缓存的逻辑“或”组成。表 3-1 对 glClear()命令进行了总结，给出了可能进行清除操作的缓存、缓存名称以及后面讨论它们的章节。

表 3-1 清空缓存

缓存	名称	参考
颜色缓存	GL_COLOR_BUFFER_BIT	第 6 章
深度缓存	GL_DEPTH_BUFFER_BIT	第 11 章
累积缓存	GL_ACCUM_BUFFER_BIT	第 11 章
模板缓存	GL_STENCIL_BUFFER_BIT	第 11 章

void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);

此函数在 RGBA 模式中，设置当前清空颜色来清空颜色缓存。关于 RGBA 模式的详细说明请见第 6 章。red, green, blue 和 alpha 的值截取在[0, 1]之间。缺省的清除颜色为黑色(0, 0, 0, 0)。

void glClear(GLbitfield mask);

以当前清除值清除指定的缓存。参数 mask 是表 3-1 所列各值按位或的连接。

在发出一个命令来清空一组缓存之前，如果程序员想要得到缺省的颜色、深度值、累积颜色及模板索引以外的特性，必须对要清空的每一个缓存进行设置。除了用 glClearColor()和 glClearDepth()命令为清空的颜色和深度设置当前值，glClearIndex(), glClearAccum()，和 glClearStencil()可以分别指定清空对应缓存的颜色索引、累积颜色和模板索引。对这些缓存的定义和使用请参见第 6 章和第 11 章。

由于要操作窗口中的每一个像素（可能是成百万的），所以清空常常是比较缓慢的，而一些图形硬件允许同时清除一系列的缓存，因此 OpenGL 允许程序员指定多个缓存。不支持同时进行清除的硬件系统，则只能顺序地进行操作。命令：

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

和
```
glClear(GL_COLOR_BUFFER_BIT);
glClear(GL_DEPTH_BUFFER_BIT);
```

尽管可以最后达到相同的效果，但在许多计算机上，前面的形式运行得要快。

3.1.2　指定颜色

在 OpenGL 中，对所要绘制物体的形状描述与对其颜色的描述是相互独立的。一个特定的几何物体不论在什么时候画出，都使用当前指定的着色方案。着色方案可以简单如“将所有的物体画成深蓝色”，也可复杂如“假定物体由海蓝色塑料制成，在某一处有一桔黄色的点光源，其他地方为红褐色普通光”。总之，OpenGL 程序员首先设置颜色和着色方案，然后才能画出物体。只要不改变颜色和着色方案，所有的物体都使用该颜色或该着色方案画出。这一追踪当前颜色的方法使 OpenGL 可达到较高的绘图性能，例如，伪代码：

```
set_current_color(red);
draw_object(A);
draw_object(B);
set_current_color(green);
set_current_color(blue);
draw_object(C);
```

将物体 A 和 B 画成红色，将物体 C 画成蓝色。第四行将当前颜色设为绿色的命令是无效的。

着色、光照和阴影都是较重要的主题，需要整章或整节的内容来讨论。然而，要画出可见的几何图元，还需要一些设置当前颜色的基本知识。在下一段我们先简单地涉及这个问题。详细的讨论请参见第 6 章和第 7 章。

要设置颜色，需使用命令 glColor3f()。它有三个参数，都是 0.0 到 1.0 之间的浮点数。这些参数按其先后顺序分别为：红色、绿色、蓝色颜色分量。也可以将这三个值看成是合成一个颜色值的分量：0.0 指不用该颜色分量，1.0 意味着使用该颜色分量的最大值。由此，代码：

```
glColor3f(1.0, 0.0, 0.0);
```

将产生系统能生成的最强的红色，而没有蓝色和绿色分量。三个值全为 0 为黑色，全为 1 对应白色。将三个分量全设为 0.5 产生灰色（白色和黑色之间的颜色）。以下为八种情况所设置的颜色：

glColor3f(0.0, 0.0, 0.0)　黑色
glColor3f(1.0, 0.0, 0.0)　红色
glColor3f(0.0, 1.0, 0.0)　绿色
glColor3f(1.0, 1.0, 0.0)　黄色
glColor3f(0.0, 0.0, 1.0)　蓝色
glColor3f(1.0, 0.0, 1.0)　洋红色
glColor3f(0.0, 1.0, 1.0)　蓝绿色
glColor3f(1.0, 1.0, 1.0)　白色

读者可能注意到涉及颜色的设置命令有四个参数。除去红、绿、蓝三个通常的颜色分量外，第四个参数是一个 alpha 值。在混合一章中将对其进行详细讨论。目前将此参数设置为 0.0。

3.1.3 完成绘图

许多现代的图形系统可以被看成是一条装配线，即图形管道。中央处理单元（CPU）发出绘图命令，可以由其他硬件来实现几何变换、裁剪，然后进行阴影和纹理操作，最后将数值写入位平面来显示（操作顺序参见附录 1）。在高端（high-end）体系结构中，每一个操作由不同的硬件来执行，这些硬件是为快速地执行特定的任务而专门设计的。在这样一个体系结构中，CPU 没有必要等待每一个绘图命令完成后再发送下一条命令。当 CPU 将一个顶点发送到图形管道中时，变换硬件正对上一个发送的顶点进行变换，而在这个顶点之前发送的顶点则正在被剪裁，其他点也是如此。在这样一个体系中，如果 CPU 等待每一条命令完成后发送下一条，性能将受到很大损害。

另外，应用程序可以在客户/服务器的网络系统上运行。当在客户计算机上运行主程序，而程序员通过网络在另外的服务器终端上观看绘图结果时，由于网络传输伴有相当严重的超前，通过网络每次只发送一条命令，其低效程度是可怕的。而且，客户的网络代码常常无法得知图形程序是否已经绘制完一帧画面或场景。在最糟的情况下，网络将无限制地等待足够的绘图命令来填满一个数据包，这样将永远无法看到完整的图画。

基于以上原因，OpenGL 提供了命令 glFlush()，强迫客户发送网络数据包，而不管其是否填满。在没有网络并且所有命令立刻都在服务器自身执行的情况下，glFlush()常常不起作用。当然，如果程序员想要编写有网络和没有网络的情况下都能运行的程序，就需要在每一个画面或场景的结尾调用 glFlush()。注意 glFlush()并不等待绘图完成，而只是强迫绘图开始执行，由此保证即使后面没有进一步的渲染命令，先前所有命令都能在有限时间内得以执行。

少数一些命令，例如双缓存模式下的交换缓存命令，在调用之前会自动地将未完成的命令送入网络中。

---

void glFlush(void);

强制前面发出的 OpenGL 命令开始执行，从而保证其在有限的时间内得以执行。

如果 glFlush()并不能满足要求，可以试用 glFinish()命令。这一命令如 glFlush()命令一样压入网络，然后等待图形硬件或网络发出在帧缓冲器内的绘图已经完成的通知。如果程序员想使任务同步，要用到 glFinish()，例如，利用 Display PostScript 在渲染图的顶部画上标签之前，要确定三维渲染图处在屏幕上；在接收使用者的输入之前，一定要保证绘图已经完成。

在发出 glFinish()命令后，绘图进程将被阻塞，直到接收到图形硬件（如果在网络上运

行，则为客户）发出绘图已经完成的通知为止。由于 glFinish()需要进行双向通讯，所以过多的使用此命令将降低应用程序的性能，尤其是在网络上运行时。所以，若使用 glFlush()已经能满足需求，那么就不要使用 glFinish()。

---

void glFinish(void);

强制前面发送的所有 OpenGL 命令完成。直到前面命令的所有效果都得到完全实现后，该命令才返回。

3.1.4　隐藏面的消除

当绘制一个由三维物体组成的场景时，某些物体可能被其他物体全部或部分地遮盖。随着视点的改变，这种遮盖关系也将随之改变。从相反的方向观看场景时，原来在某物体前面的物体现在都在此物体后面了。为了绘出真实的场景，必须保持这种遮盖关系。如果程序代码如下所示：

```
while (1)
  get_viewing_point_from_mouse_position();
  glClear(GL_COLOR_BUFFER_BIT);
  draw_3d_object_A();
  draw_3d_object_B();
}
```

则可能发生这种情况：对于一些视点位置，物体 A 遮住了物体 B; 而对另一些位置，遮盖关系则可能相反。如果没有其他操作，在前的代码总是先画，物体 B 总是后画出，因此物体 B 总在物体 A 之上，与如何选择观察位置无关。

消除实体中被其他物体遮挡的部分称为隐藏面的消除（隐藏线的消除，是消除物体的线框图中被遮盖的部分，在此处不作讨论。详见隐藏线的消除）。实现隐藏面的消除的最简单的办法是用深度缓存（有时叫作 z 缓存）。

深度缓存的实现是通过将距视点的深度或距离与窗口上的每一个像素联系起来进行的。初始情况下，所有像素的深度值通过参数为 GL_DEPTH_BUFFER_BIT 的 glClear()命令设置成可能的最大距离，然后按任意顺序画出场景中的物体。

硬件或软件中的图形计算将绘制出的各个表面变换成窗口中的一系列像素，如果表面未被其他物体遮盖，就会现在窗口中。此外，还需要计算相对眼睛的距离。激活深度缓存时，在绘制每一个像素之前，要将已经存储在像素上的深度值与计算得到的值进行比较。如果新的像素点离眼睛比原来的像素点近的话，新的像素点的颜色和深度值就会取代那些当前写在像素中的值。如果离眼睛的距离远了，则新的像素点将被挡住，因此变化后像素点的颜色和深度信息将被忽略。采取隐藏面的消除，将部分信息忽略而不用于绘图，这样程序的性能将

大大提高。

使用深度缓存前，首先必须激活它。这一步只需进行一次。每画一次场景，在画之前需要清空深度缓存，然后再以任意的顺序在场景中画出物体。

将前面的程序片段变换成能够执行隐藏表面消除的功能，可以修改如下：

```
glEnable(GL_DEPTH_TEST);
...
while (1)
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   get_viewing_point_from_mouse_position();
   draw_3d_object_A();
   draw_3d_object_B(); }
```

glClear()所带的参数同时清除了深度和颜色缓存。

3.2 点、线和多边形的描述

这一节介绍了如何描述 OpenGL 的几何图元。所有的几何图元最终都以其顶点坐标来描述。这些坐标定义了点本身、线段的端点或多边形的角点。下一节讨论这些图元的显示以及在其显示过程中如何进行控制。

3.2.1 点、线和多边形的定义

读者对于数学上的点、线和多边形等术语一定有相当清楚的概念。但这些术语在 OpenGL 中的意义并不完全相同。理解这种差别是十分必要的。差别的产生主要是因为，数学家可以在几何学的完备世界中考虑问题，而进行仿真的 OpenGL 则必须考虑真实世界的种种限制。

例如，一个差别是来自计算机能力的限制。在任何 OpenGL 应用中，浮点数的计算精度是有限的，存在着舍入误差。因此，OpenGL 中的点、线和多边形的坐标都要受到影响。

另一种差别源于以位图方式进行的图形显示。在这样的显示中，最小的显示单位是一个像素，尽管像素宽度可以小于百分之一厘米，但相对于数学中的无限小（指点）和无限窄（指线），这仍然是太大了。OpenGL 在计算中，假设点由浮点数向量来表示。然而在实际绘制中点经常（但不总是）就是单个像素。因此 OpenGL 将许多个坐标值相差不多的点都绘制在同一个像素点上。

1. 点

一个点是由一系列定义为顶点的浮点数来表示的。内部计算中总认为顶点是三维的。由用户定义的二维顶点（也就是只有 X 坐标和 Y 坐标）均由 OpenGL 将其 Z 坐标缺省赋值为零。

OpenGL 是在三维投影几何学中的齐次坐标下进行工作的，因此对于内部计算，所有的

顶点都由四个浮点坐标（x, y, z, w）来表示。如果 w 不为 0，这些坐标对应于欧氏几何中的三维空间点（x/w, y/w, z/w）。在 OpenGL 命令中可以定义 w 坐标，但是事实上很少这样做。如果 w 坐标没有指定，就认为它是 1.0。对于齐次坐标系更详细的知识，请参见附录 7。

2. 线

在 OpenGL 中，线指的是线段，而不是数学意义上的在两个方向上伸向无穷远的直线。定义一系列相连的、甚至是闭合的线段有一些简单的办法（如图 3-1 所示）。虽然如此，无论是何种情况，一组线段相连而成的线仍是以这些线段端点所处的顶点来定义的。

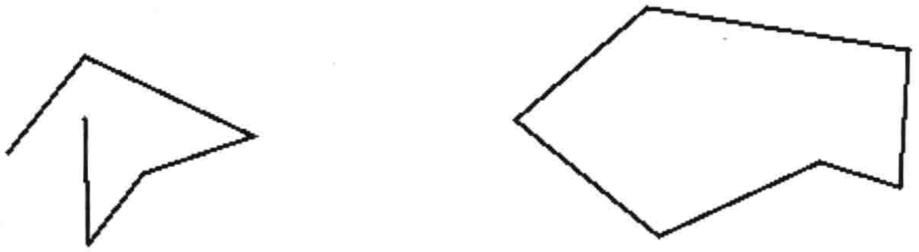
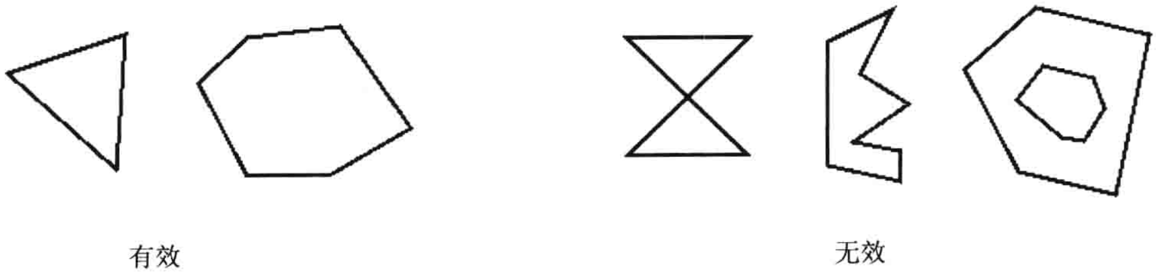


图 3-1　两类连接的线段系列

3. 多边形

多边形是指由线段组成的单个闭合回路围成的区域，这些线段由其端点处的顶点指定。多边形常常通过填充其内部的像素来画出，但也可以将其画成轮廓线或一些点，见多边形详解。

一般而言，多边形可能很复杂，所以 OpenGL 对基本多边形的组成做了一些强制限定。首先，OpenGL 多边形的边不可以相交（数学上称其为简单多边形）。第二，OpenGL 多边形必须是凸的，这意味着多边形不能是锯齿状的。确切地说，如果给定两个多边形内部的点，连接这两个点的线也在多边形内部，则称这个区域是凸的。图 3-2 给出了一些有效及无效的 OpenGL 多边形例子。当然，OpenGL 并不对组成多边形边界的线段数量作限制。注意，有洞的多边形是无法描述的。这种多边形不能由单闭合回路组成的边界来描述，是非凸的。注意，如果程序员为 OpenGL 指定一个非凸的填充多边形，OpenGL 将绘制出意想不到的图形。例如，在许多系统中，只有多边形的凸体被填充，但在另一些系统中，填充的可能不只是凸体。



有效　　　　无效

图 3-2　有效及无效的多边形

对于许多应用程序来说，要用到非简单多边形、非凸多边形或有洞的多边形。这些多边形可以由简单的凸多边形组合而成，在 GLU 中给出了一些描述更复杂物体的函数。这些子函数通过更为复杂的处理，对这些多边形进行网格化，即将它们分解成一组一组简单的、能够进行渲染的 OpenGL 多边形。详细的网格化子函数请参见附录 3。OpenGL 之所以对有效的多边形类型作出限制，是为了可以更方便地提供进行多边形快速渲染的硬件。

由于 OpenGL 顶点一般是三维的，这些组成特定多边形边界的点并不必位于空间的同一平面内（当然，在许多情况下它们是在同一平面内的——例如如果所有的 Z 坐标为 0，或是多边形为一个三角形时）。如果一个多边形的顶点并不处在同一平面内，经过空间的一系列旋转、视点的改变和向显示屏幕的投影后，这些点将不再组成一个简单的凸多边形。例如，假想一个四边形，四个点稍稍偏离了原平面，沿边缘处去观察将看到如图 3-3 所示类似一张弓弦的非凸多边形，这个多边形 OpenGL 无法保证能被正确地渲染。如果利用由真实表面上的点组成的四边形来逼近这个表面时，这种情况是常常出现的。一般的，可以通过利用三角形逼近来避免这种情况——因为所有的三角形的顶点都处在一个平面上。

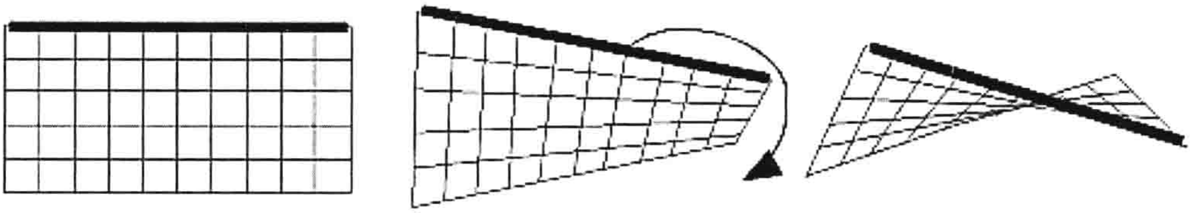


图 3-3 非平面多边形变换成非简单多边形

4. 矩形

由于矩形在图形应用中是非常普遍的，因此 OpenGL 提供了一种填充矩形的绘制函数：glRect*()。矩形当然可以当作一个多边形来绘制，但是特定的 glRect*()命令将使特定的 OpenGL 实现得到大大的优化。

---

void glRect{sifd}(TYPEx1, TYPEy1, TYPEx2, TYPEy2);
void glRect{sifd}v(TYPE*v1, TYPE*v2);

此函数绘出由对角顶点(x1, y1) 和 (x2, y2)定义的矩形。该矩形在平面 z=0 上，其边平行于 X 轴和 Y 轴。如果以向量形式表示函数，则顶点由两个指向数组的指针给出，每个数组包括一个(x, y)数对。

注意：尽管在三维空间里矩形最初有一个特定方向（在平面 x-y 上且平行于坐标轴），程序员仍可以通过旋转或其他变换，对矩形方向进行改变。对此操作的详细介绍请参见第 4 章。

5. 曲线

所有的光滑曲线和曲面都可以由短线段和小多边形区域以任意的精度进行逼近。因此，只要将曲线和曲面划分得足够细，然后用直线段和平面多边形近似便可以使它们看起来是弯曲的（如图 3-4 所示）。如果要求得比较严格的话，例如将线段和平面多边形分割成比屏幕上的一个像素还小的程度，那么这种近似将是十分有效的。

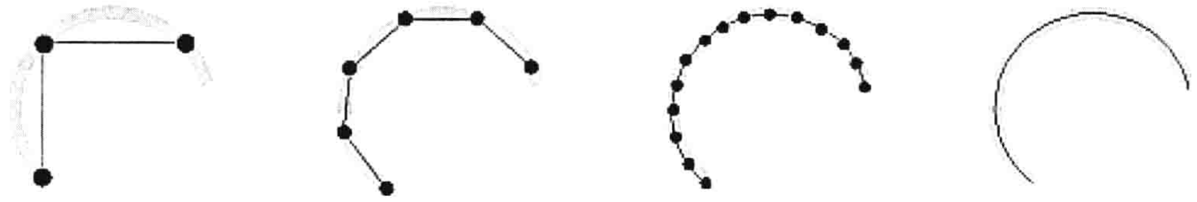


图 3-4 曲线的逼近

尽管曲线（面）并不是基本图元，OpenGL 仍然提供了一些直接绘制的办法。关于如何绘制曲线和曲面请参见第 12 章。

3.2.2 定义顶点

在 OpenGL 中，所有的几何物体最终都要描述成一个顶点的有序集合。程序员可以用 glVertex*()命令来定义一个顶点。

---

void glVertex{234}{sifd}[v](TYPEcoords);

此函数定义用于描述几何物体的一个顶点。通过选择合适的命令，程序员可以用多到四个坐标（x，y，z，w）、少到两个坐标（x，y）来指定一个特定的顶点。如果选择了一种没有明确指定 z 或 w 的命令，则 OpenGL 认为 z 为 0 而 w 为 1。执行 glVertex*() 调用应在 glBegin() 和 glEnd()成对命令之间进行。

以下是一些使用 glVertex*()的例子：

```
glVertex2s(2, 3);
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex4f(2.3, 1.0, -2.2, 2.0);
GLdouble dvect[3] = {5.0, 9.0, 1992.0};
glVertex3dv(dvect);
```

第一个例子表示一个三维坐标为（2，3，0）的顶点（z 未指定，就认为是 0）。在第二个例子中的坐标为（0.0，0.0，3.1415926535898）（双精度浮点型数）。第三个例子表示一个三维坐标为（1.15，0.5，-1.1）的顶点（这里的 X，Y，Z 坐标是经过 W 坐标处的结果）。在最后一个例子里，dvect 是一个指向有三个双精度浮点数的数组的指针。

对于某些计算机，glVertex*()的向量形式是更加有效的，因为只有一个参数需要传送到

图形子系统中，这样，特定的硬件便可以在一个批处理中传送所有的坐标。对于这样工作的计算机，程序员可以将数据进行排列，使顶点坐标顺序地存入内存中，这对提高应用程序性能是十分有利的。

3.2.3 OpenGL 的几何绘图图元

现在我们已经知道如何指定顶点，但仍需要了解如何通知 OpenGL 根据这些顶点创建一系列的点、线或一个多边形。为了实现上述过程，程序员需要将每组顶点放在 glBegin()调用和 glEnd()调用之间，由传递给 glBegin()命令的参数来根据这些顶点创建哪种类型的几何图元。例如，下面的代码指定了绘出如图 3-5 所示多边形的顶点。

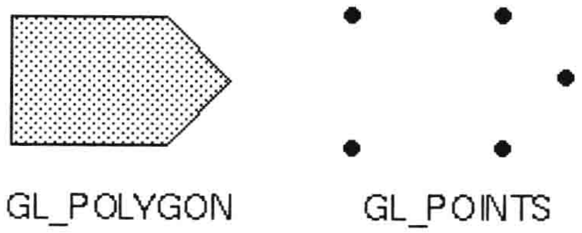


图 3-5 绘制一个多边形或点集

```
glBegin(GL_POLYGON);
   glVertex2f(0.0, 0.0);
   glVertex2f(0.0, 3.0);
   glVertex2f(3.0, 3.0);
   glVertex2f(4.0, 1.5);
   glVertex2f(3.0, 0.0);
glEnd();
```

如果用参数 GL_POINTS 代替参数 GL_POLYGON，则图元就是如图 3-5 所示的五个简单的点。在表 3-2 的功能总结中列出了 10 个 glBegin()函数可能取到的参数及相应的图元类型。

---

void glBegin(GLenum mode);

此函数标志着描述一个几何图元的顶点列表的开始。图元的类型由 mode 来给定，可以取表 3-2 中的任何值:

---

void glEnd(void);

此函数标志着顶点列表的结束。

表 3-2　几何图元的名称和意义

值	意义
GL_POINTS	独立的点
GL_LINES	一对顶点表示一条线段
GL_POLYGON	简单的、凸多边形边界
GL_TRIANGLES	表示三角形顶点的三元组
GL_QUADS	表示四边形顶点的四元组
GL_LINE_STRIP	一系列的相连的线段
GL_LINE_LOOP	同上，在开始和结束顶点之间增加一条线段，使其闭合
GL_TRIANGLE_STRIP	连接的三角形条带
GL_TRIANGLE_FAN	连接的三角形扇面
GL_QUAD_STRIP	连接的四边形条带

图 3-6 给出了表 3-2 中所列出的几何图元的实例。下面给出用于绘制各个物体的所有像素的精确描述。注意，在点以外，还定义了其他一些类型的线和多边形。显然，画同一个几何图元可以有许多办法。程序员如何选择这些方法取决于顶点数据。

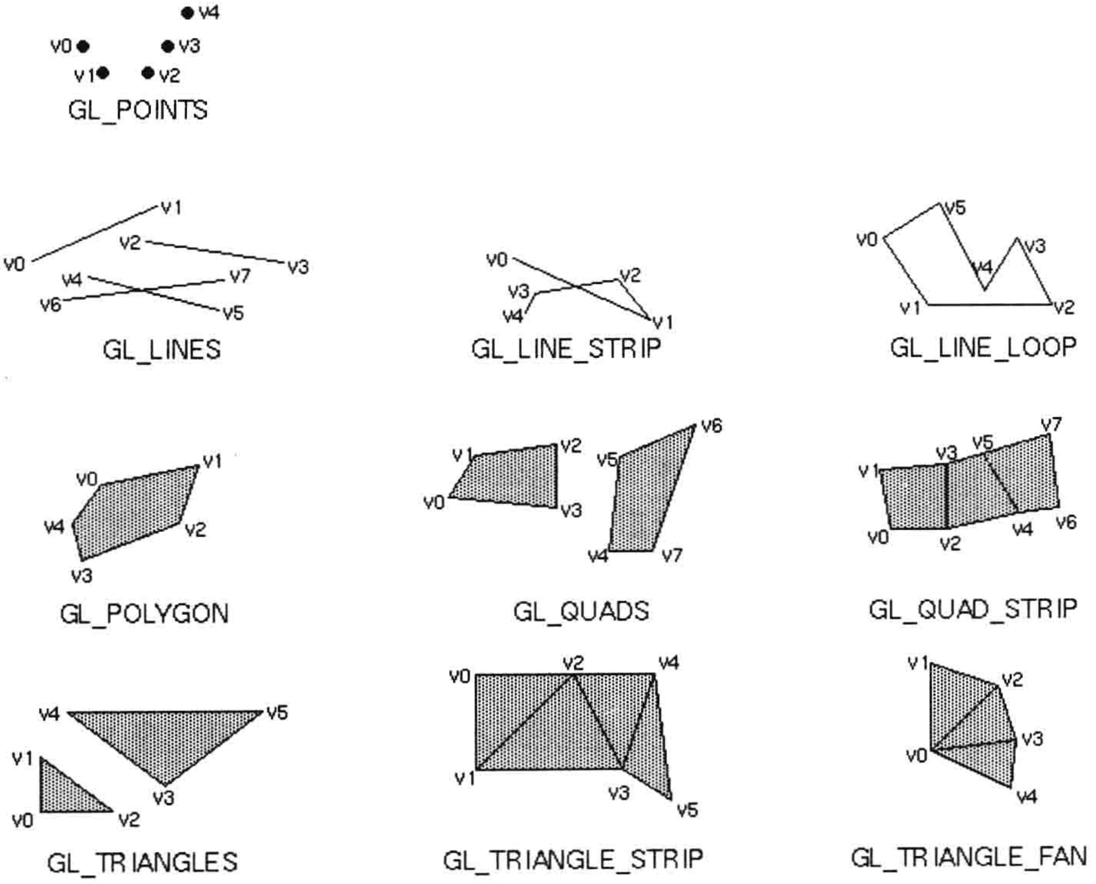


图 3-6　几何图元类型

在下面的描述中，我们假设在 glBegin()和 glEnd()命令对之间描述了 n 个顶点（v0, v1, v2, ... , vn-1）。

GL_POINTS	在每一个顶点处画一个点
GL_LINES	画出多条线段。线段画在 v0 和 v1 间, v2 和 v3 间，以此类推。如果 n 为奇数，最后的一条线段画在 vn-3 和 vn-2 之间,而顶点 vn-1 将被忽略。
GL_POLYGON	以点 v0, ... , vn-1 为顶点划出一个多边形。n 必须大于等于 3，否则将什么也画不出来。另外，指定的多边形线段必须不与自己相交，必须是凸的。如果这些顶点不满足上述要求，将产生不可预料的结果。
GL_TRIANGLES	以 3 个顶点为一组绘制三角形。如顶点 v0, v1, v2 为一组，然后是 v3, v4, v5，以此类推，绘出一系列的三角形。如果 n 不能被 3 整除，最后的一个或两个顶点将被略去不计。
GL_LINE_STRIP	以 2 个顶点为一组绘制多条线段。如从 v0 到 v1，然后是 v1 到 v2，以此类推，最后由 vn-2 到 vn-1 画出多条线段。由此，共有 n-1 条线段。n 必须大于 1 才能绘制出线段。在描述线段带（或闭合线段）时没有特殊的限制。这些线可以自由相交。
GL_LINE_LOOP	同 GL_LINE_STRIP，只是最后一条线段由 n-1 画向 v0，从而使多条线段闭合。
GL_QUADS	以 4 个顶点为一组，绘制多个四边形。如以顶点 v0, v1, v2, v3 为一组，然后以 v4, v5, v6, v7 为一组，以此类推，绘制一系列的四边形。如果 n 不是 4 的整数倍，最后的一个、两个或三个点将被略去不计。
GL_QUAD_STRIP	画出一系列相连的四边形（四条边的多边形），以 v0, v1, v3, v2 开始；然后是 v2, v3, v5, v4； 接下来是 v4, v5, v7, v6，以此类推，见图 3-6。n 必须大于 4，否则什么也画不出来。如果 n 为奇数，最后一个顶点忽略。
GL_TRIANGLE_STRIP	绘制一系列相连的三角形。先以顶点 v0, v1, v2 绘制，然后是 v2, v1, v3（注意顺序），接下来是 v2, v3, v4，以此类推。在这里要求按一定的顺序绘制，是保证三角形都以同样的方向画出，以便三角形带可组成表面的一部分。图 3-6 说明了按顺序绘制的意义。同样，n 必须大于 3。
GL_TRIANGLE_FAN	与 GL_TRIANGLE_STRIP 相同，但是顶点为 v0, v1, v2，然后是 v0, v2, v3，接着是 v0, v3, v4 等等。参见图 3-6。

3.2.4　使用 glBegin()和 glEnd()的限制

顶点所载的最重要的信息是它们的坐标，由命令 glVertex*()指定。程序员可以利用特定的命令为每个顶点提供附加的顶点数据——颜色、法线向量、纹理坐标，或是这些数据的任意组合。其他一些命令出现在 glBegin()和 glEnd()之间也是有效的。表 3-3 即为这些用在 glBegin()和 glEnd()命令对之间的有效命令的完整的列表。

表 3-3　在 glBegin() 和 glEnd()之间的有效命令

命令	命令目的	参考
glVertex*()	设置顶点坐标	第 3 章
glColor*()	设置当前颜色	第 6 章
glIndex*()	设置当前颜色索引	第 6 章
glNormal*()	设置法线向量坐标	第 3 章
glEvalCoord*()	生成坐标	第 12 章
glCallList(),glCallLists()	执行显示列表	第 5 章
glTexCoord*()	设置纹理坐标	第 10 章
glEdgeFlag*()	控制边的绘制	第 3 章
glMaterial*()	设置材料属性	第七章

其他任何 OpenGL 命令，出现在 glBegin()和 glEnd()命令对之间都将是无效的，也就是说，调用时将产生错误。然而要注意，这个限制只是对 OpenGL 命令，程序员可以随意在其中加入其他程序语言模块。例如下列代码绘制的是一个圆的轮廓：

```
#define PI 3.1415926535897;
GLint circle_points = 100;
glBegin(GL_LINE_LOOP);
for (i = 0; i < circle_points; i++)
  angle = 2*PI*i/circle_points;
  glVertex2f(cos(angle), sin(angle));
}
glEnd();
```

注意：本例并不是绘制圆的最有效的方法，尤其是要反复进行时。实际的图形命令应用起来是十分快捷的，但此代码中对每一个顶点的角度都进行了计算并调用了 sin()和 cos()子程序；另外还有一个循环计算。如果需要绘制许多圆，应该一次性计算出它们的顶点并存储在数组里，再创建一个显示列表（见第 5 章）或用 GLU 子函数（见附录 3）来绘制它们。

除了把 glVertex*()命令编辑在显示列表中以外，平时该命令必须出现在 glBegin()和 glEnd()命令对之间。在其他地方出现将不能完成任何操作。即使出现在显示列表中，也只

是出现在 glBegin()和 glEnd()命令对时才能得以执行。

尽管在 glBegin()和 glEnd()之间允许使用许多命令，但只有调用了 glVertex*()命令时才能生成顶点。调用该命令时，OpenGL 将给顶点分配当前的颜色、纹理坐标、法线向量信息等等。下面的代码清楚地表明了这一点。无论有无其他颜色命令，程序都将以红色绘制第一个点，以蓝色绘制第二及第三个点。

```
glBegin(GL_POINTS);
    glColor3f(0.0, 1.0, 0.0);                    /* 绿色  */
    glColor3f(1.0, 0.0, 0.0);                    /* 红色 */
    glVertex(...);
    glColor3f(1.0, 1.0, 0.0);                    /* 黄色 */
    glColor3f(0.0, 0.0, 1.0);                    /* 蓝色 */
    glVertex(...);
    glVertex(...);
glEnd();
```

程序员可以在 lBegin()和 glEnd()之间使用 glVertex*()命令的 24 种形式的任何组合。但在实际应用中，每个特定场合常常采取一种形式的调用。

3.3 显示点、线和多边形

缺省情况下，点在屏幕上是以单个像素画出的，线是实线、一个像素宽，多边形是实心且填充的。下面一段讨论了如何改变这些缺省显示模式的一些细节问题。

3.3.1 关于点的细节

为了控制渲染的点的尺寸，应使用 glPointSize()命令，参数为期望尺寸，以像素为单位。

void glPointSize(GLfloat size);

设置渲染点的像素宽度。size 必须大于 0.0，缺省为 1.0。

对于不同大小的点，屏幕上实际画出的像素集合依赖于是否激活了锯齿现象的消除模式（锯齿现象的消除是一种用于渲染点或线的技术。详见“锯齿现象的消除”）。如果锯齿现象的消除未被激活（为缺省情况），小数的宽度将经取舍化为整数的宽度，然后画出像素在屏幕上的方形排列区域。因此，如果宽度为 1.0，此方形为 1×1 像素，如果宽度为 2.0，此方形为 2×2 像素，依此类推。

如果锯齿现象消除被激活，将以圆形排列画出一组像素，并把边界上的像素画得暗一些，这样一来使边缘显得较为光滑。在这种模式下，对非整型宽度进行了取舍。

绝大多数的 OpenGL 实现支持非常大的点的尺寸。然而在特定实现中，要将未经锯齿现象消除的点的尺寸经过取舍后近似为整数值，限定在锯齿现象消除点的最大尺寸以内。程序员可以用带有参数 GL_POINT_SIZE_RANGE 的 glGetFloatv()命令来获得此浮点值。

3.3.2　关于线的细节

在 OpenGL 中，可以指定任意宽度的线，并可以画成不同的方式，如点线、虚线以及点划线等等。

1. 宽度线

---

void glLineWidth(GLfloat width);

以像素为单位设置线宽；width 的值必须大于 0.0，缺省情况下为 1.0。

实际的线渲染受到锯齿现象消除模式的影响，这点和渲染相同（参见“锯齿现象的消除”）。没有锯齿现象的消除操作，宽度为 1, 2, 3 以一个、两个、三个像素为宽度画线。在锯齿现象消除操作激活情况下，有可能出现非整型的线宽，而且边缘处的像素点常常是被部分地填充。同点的尺寸一样，一个特定的 OpenGL 实现将未经锯齿现象消除线的宽度，近似成最接近的整型数后，限制到最大的锯齿现象消除线的宽度以内。程序员可以用带有参数 GL_LINE_WIDTH_RANGE 的 glGetFloatv()命令来获得此浮点值。

注意：在缺省情况下线是一个像素宽，这样一来同样的线在较低分辨率的窗口下将显得较宽。对于计算机的显示，这常常不成为什么问题。但是若使用 OpenGL 将图形渲染到一台高分辨率的绘图仪上，一个像素的线可能几乎什么也看不到了。为了得到与分辨率无关的线宽，程序员需要考虑像素的物理尺寸。

对于经过了锯齿现象消除的宽度线，线宽并不是垂直于线来测量的。如果斜率的绝对值小于 1.0，则沿 y 方向来测量，否则，沿 x 方向测量。渲染一条消除锯齿现象的线完全等同于渲染一个填充的给定宽度，中心即为精确线的矩形。对于渲染填充的多边形区域的讨论，参见“多边形详解”一节。

2. 点划线

为了得到点划线（点线或虚线），需要用到 glLineStipple()命令来定义点划模式，然后用 glEnable()激活线点划模式。

```
glLineStipple(1, 0x3F07);
glEnable(GL_LINE_STIPPLE);
```

---

void glLineStipple(GLint factor, GLushort pattern);

设置当前的线的点划模式。参数 pattern 是由多个 0 和 1 组成的 16 位数，并在需要时重复点划一条给定的线。1 表示基于每个像素、从 pattern 的低位开始进行绘制，0 表示不绘制。用 factor 去乘以每一个连续的 1、0 序列，便可以对 pattern 进行扩展。因此，如果在 pattern 中连续出现三个 1，factor 为 2 则意味着扩展到六个 1。factor 的值需要限制在 1 到 255 之间。线的点划模式必须以参数 GL_LINE_STIPPLE 调用 glEnable()命令后才被激活。将同该参数传给 glDisable()则取消线的点划模式。

在前面的例子中，pattern 为 0x3F07（如果写成二进制，则为 0011111100000111）时，则将以“先绘制 3 个像素，隔 5 个像素再绘制 6 个像素、然后再隔 2 个像素”这种方式画出一条线（记住，低位在前）。如果 factor 为 2，pattern 将扩展为“先绘制 6 个像素，隔 10 个像素再绘制 12 个像素，然后再隔 4 个像素”。图 3-7 显示了按不同的 pattern 和重复因子 factor 画出的线。如果不激活线的点划模式，将以 pattern 为 0xFFFF 和 factor 为 1 继续进行绘制（用带有参数 GL_LINE_STIPPLE 的函数 glDisable()将取消点划模式），注意点划线可以与宽度线结合起来产生宽度点划线。

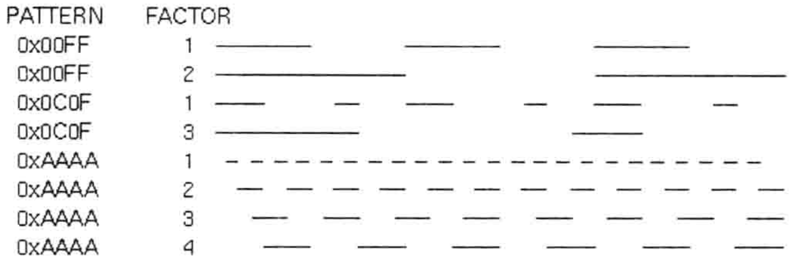


图 3-7　点划线

线的点划问题可以这样来考虑：绘制线段时，pattern 是一位一位变换的，每次只绘制一个像素（若 factor 不为 1，则绘制 factor 个像素）。当在 glBegin()和 glEnd()之间绘出一系列的连接线段时，每画完一条线段再画下一条线段时，pattern 都要持续地变换。通过这种办法，点划模式将持续作用于一系列连续线段。执行 glEnd()时，将被重新设置 pattern。这样一来，假如在取消点划模式之前还需要绘制多条线段，点划将在 pattern 的开头重新开始。若绘图采用参数 GL_LINES，则每条独立的线绘制时，pattern 都将重新设置。例 3-1 显示了用两种不同点划模式画出的有宽度的线。同时该例也说明了，如果线不是作为连接的线带而是作为一系列的单独的线段画出时将会发生的现象。运行此程序的结果如图 3-8 所示。

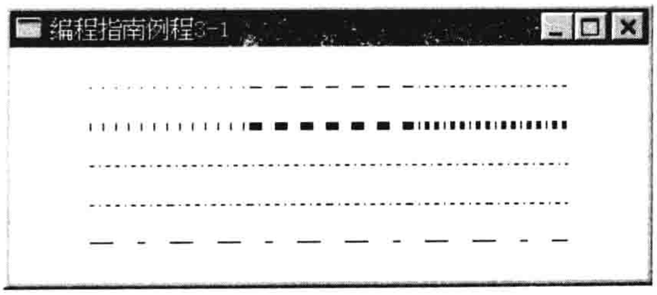


图 3-8　具有宽度的点划线

例 3-1　使用线的点划模式

```
#include <windows.h>

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>


void drawLine(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2)
{
     glBegin(GL_LINES);
         glVertex2f (x1,y1);
         glVertex2f (x2,y2);
    glEnd();
}

void myinit (void)
{
    /*  将背景清为白色  */
   glClearColor (1.0, 1.0, 1.0, 0.0);
   glShadeModel (GL_FLAT); //无明暗差别的阴影
}

void CALLBACK display(void) //注意定义回调
{
   int i;
   glClear (GL_COLOR_BUFFER_BIT);
/*  将所有的线画为黑色  */
   glColor3f (0.0, 0.0, .0);

/*  在第一行，以不同的点划方式画出 3 条线，  */
   glEnable (GL_LINE_STIPPLE);
   glLineStipple (1, 0x0101);              /*点线*/
   drawLine (50.0, 125.0, 150.0, 125.0);
   glLineStipple (1, 0x00FF);              /*  虚线*/
```

```
    drawLine (150.0, 125.0, 250.0, 125.0);
    glLineStipple (1, 0x1C47);              /*  点划线  */
    drawLine (250.0, 125.0, 350.0, 125.0);

/*在第二行，以不同点划方式画出三条宽度线 */
    glLineWidth (5.0);
    glLineStipple (1, 0x0101);
    drawLine (50.0, 100.0, 150.0, 100.0);
    glLineStipple (1, 0x00FF);
    drawLine (150.0, 100.0, 250.0, 100.0);
    glLineStipple (1, 0x1C47);
    drawLine (250.0, 100.0, 350.0, 100.0);
    glLineWidth (1.0);

/*  在第三行，都以点划线模式画出 6 条线 */
/*  为连接线带的一部分 */
    glLineStipple (1, 0x1C47);
    glBegin (GL_LINE_STRIP);
    for (i = 0; i < 7; i++)
    glVertex2f (50.0 + ((GLfloat) i * 50.0), 75.0);
    glEnd ();

/*  在第四行，以点划线画出 6 条独立的线 */
    for (i = 0; i < 6; i++)
     {
        drawLine (50.0 + ((GLfloat) i * 50.0),
           50.0, 50.0 + ((GLfloat)(i+1) * 50.0), 50.0);
    }

/*  在第五行，画一条点划线，重复因子为 5*/
    glLineStipple (5, 0x1C47);
    drawLine (50.0, 25.0, 350.0, 25.0);
    glFlush ();
}
```

```
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 400, 150);
    auxInitWindow ("编程指南例程 3-1");
    myinit ();
    auxMainLoop(display);
     return 0;
}
```

3.3.3 关于多边形的细节

多边形常常是通过填充包围在边界以内的所有像素来绘制，也可以将其绘制成轮廓，或者干脆只简单地画出其顶点。一个填充的多边形必须是以实体填充，或者用特定的点划模式来绘制。需要指出的是，在绘制多边形时，相邻多边形的公共顶点或公共边其构成像素实际上只画一次，并且仅包含在一个多边形中。这样做使得避免了透明的多边形边被画两次而使其比其他的边显得黑一些（或亮一些，这取决于绘图的颜色）。但同时要注意，此时可能某些狭窄的多边形有一行（或列）或多行像素未被填充。多边形锯齿现象的消除远比点和线复杂。详见“锯齿现象的消除”一节。

1. 以顶点、轮廓线和实体表示的多边形

一个多边形有两个面——正面和背面，根据哪一面面向观察者进行不同的渲染。这样就允许你观察到实体的剖视图，在剖视图中，切面的内部与外部的区别是非常明显的。缺省情况下，正面和背面都以同样的方式画出。为了改变这一点，或者只想绘制轮廓或顶点，则用 glPolygonMode()命令。

---

void glPolygonMode(GLenum face, GLenum mode);

控制多边形绘制模式是正面还是背面。参数 face 可取 GL_FRONT_AND_BACK、GL_FRONT 或 GL_BACK，参数 mode 可取 GL_POINT、GL_LINE 或 GL_FILL 来指示出多边形是以点、轮廓还是填充的形式画出。缺省情况下，正面和背面都画成填充的。

例如，程序员可以两次调用该函数按填充模式绘制正面，而将背面画成轮廓线：

```
glPolygonMode(GL_FRONT, GL_FILL);
glPolygonMode(GL_BACK, GL_LINE);
```

关于如何控制将表面处理成正面还是背面的详细信息请参见下一节。

2. 多边形表面的拣选

按惯例，屏幕上的顶点以逆时针顺序出现的多边形面称为正面。程序员可以创建任意“合

理的”实体表面，在数学上这些表面称为有向拓扑面——来自方向一致的多边形。(球体、环形、茶壶是有向的，Klein 瓶则不是)。用另一句话说，程序员可以使用所有的顺时针多边形，或者是逆时针多边形（这基本上就是有向的数学定义）。

假设程序员一致地定义了一个有向表面模型，但是恰好外侧为顺时针方向。给定正面多边形的期望方向后，程序员可以用函数 glFrontFace()来切换 OpenGL 认定的多边形正面与背面。

---

void glFrontFace(GLenum mode);

控制如何确定正面多边形。缺省情况下，mode 为 GL_CCW，对应于在窗口坐标下一个投影多边形的有序顶点形成的逆时针方向。如果 mode 为 GL_CW，认为顺时针方向的表面为正面。

在许多技术术语中，确定多边形的面为正面还是背面有赖于窗口坐标下计算的多边形面积。计算这一面积可以采取这样的办法：若 n 顶点多边形的第 i 个顶点的 x、y 窗口坐标为 x_i 和 y_i，并且：

$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_i y_{i\oplus 1} - x_{i\oplus 1} y_i$$

若指定了 GL_CCW，如果 a>0，对应的多边形将被看成是正面的顶点，否则，认为是背面。若指定的是 GL_CW ，且 a<0，则对应多边形是正面，否则是背面。

由方向一致的多边形所构成的完全封闭表面中，所有的背面都是不可见的——因为它们总是被多边形正面遮挡。在这种情况下，OpenGL 一旦确定其为背面，便将其舍弃，从而可以优化绘图速度。类似地，如果观察者处在物体内部，则只有背面是可见的。可以利用 glCullFace()命令并调用 glEnable()函数激活检选（culling）模式，通知 OpenGL 舍弃正面或背面多边形。

---

void glCullFace(GLenum mode);

指出在变换到屏幕坐标之前，舍弃哪面的多边形。参数 mode 可以为 GL_FRONT、GL_BACK 或 GL_FRONT_AND_BACK，分别表示舍弃正面、背面或是全部的多边形。为了使其生效，须用带有 GL_CULL_FACE 参数的 glEnable()命令来激活“culling”模式；带有同样参数的 glDisable()命令使“culling”失效。

3. 点划多边形

缺省情况下，填充多边形是以实体模式画出的。它们也可以用 32 位乘 32 位的窗口排列点划模式来填充。程序员可以用 glPolygonStipple()命令来定义这种模式。

---

void glPolygonStipple(const GLubyte *mask);

此函数为填充多边形定义当前点划模式。参数 mask 为一个指向 32×32 位图的指针，可以理解为由多位 0 和 1 组成的掩码。如果出现 1，则绘出多边形的对应像素，如果是 0，则什么也不画。图 3-10 显示了如何利用 mask 中的特征创建点划模式。多边形点划可以用参数为 GL_POLYGON_STIPPLE 的 glEnable()和 glDisable()命令来激活或中止。mask 中数据的理解受带有 GL_UNPACK*参数的 glPixelStore*() 命令影响。请参见“控制像素存储模式”一节。

在定义当前的多边形点划模式的同时，需要程序员激活它。

```
glEnable(GL_POLYGON_STIPPLE);
```

带有同样参数的 glDisable()命令可以取消点划模式。图 3-9 显示了不带点划的多边形，以及另外两种不同点划模式的多边形。例 3-2 显示了程序的源代码。由于程序使用图 3-9 中的模式为模板在黑色背景下以白色来绘画，因此发生了白色和黑色的反转（见图 3-9 和图 3-10）。

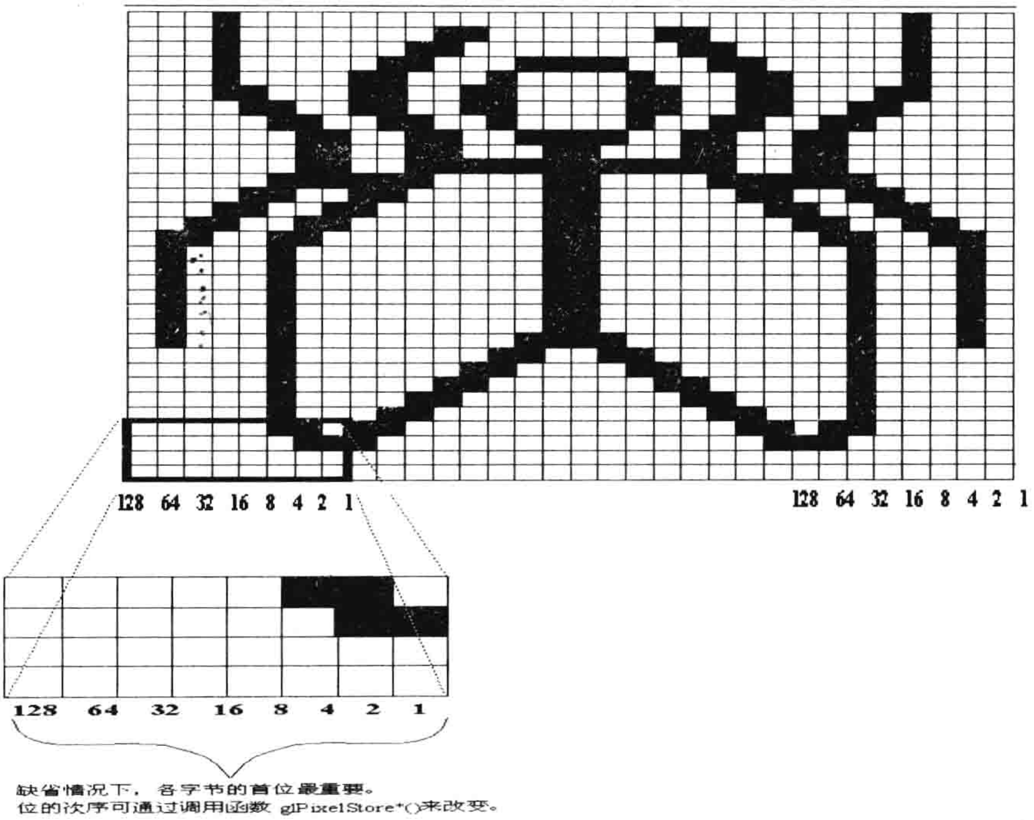


图 3-9　点划多边形

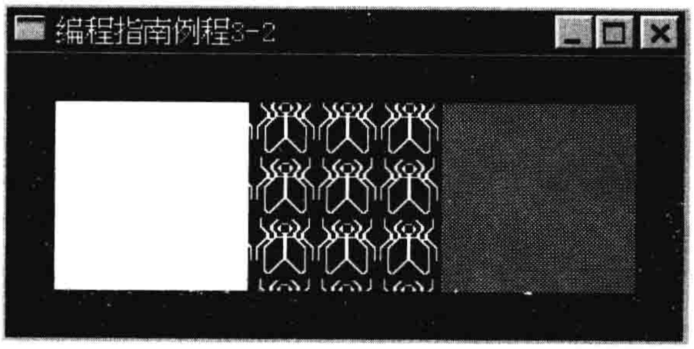


图 3-10 创建多边形点划模式

例 3-2 多边形点划模式的使用

```
#include <windows.h>

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void CALLBACK display(void)
{
   GLubyte fly[] = {
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
      0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
      0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
      0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
      0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
      0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
      0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
      0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
      0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
      0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
      0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08};
```

```
    GLubyte halftone[] = {
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55};
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glRectf (25.0, 25.0, 125.0, 125.0);
    glEnable (GL_POLYGON_STIPPLE);
    glPolygonStipple (fly);
    glRectf (125.0, 25.0, 225.0, 125.0);
    glPolygonStipple (halftone);
    glRectf (225.0, 25.0, 325.0, 125.0);
    glDisable (GL_POLYGON_STIPPLE);
    glFlush ();
}

void myinit (void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);
}
```

```
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 350, 150);
    auxInitWindow ("编程指南例程 3-2");
    myinit ();
    auxMainLoop(display);
     return 0;
}
```

正如在“显示列表设计准则”中提及的那样，读者此时可能更想利用显示列表来存储点划模式以增加效率。

4. 标识多边形边界

OpenGL 只能渲染凸多边形，但是在实践中常会产生许多非凸多边形。为了绘出这些非凸多边形，程序员常可以将其分解为一些凸多边形——一般为三角形，如图 3-11 所示——然后绘制出这些三角形。但是，如果将一般的多边形分解为三角形然后再绘制这些三角形，再采用 glPolygonMode()来绘制多边形的轮廓线，就只能得到多边形中的所有三角形的轮廓。为了解决这一问题，程序员可以通知 OpenGL 哪个顶点在边界线上，哪个顶点不在。然后，以 GL_LINE 模式绘制多边形，不画那些非边界的边。在图 3-11 中，虚线代表附加的不需画出的边。

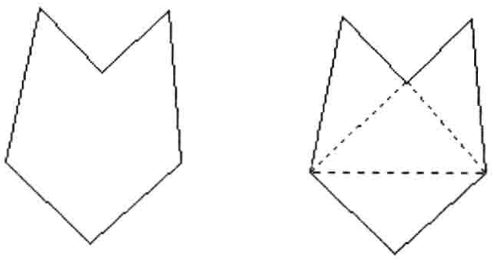


图 3-11 非凸多边形的分割

缺省情况下，所有的顶点都标识为边界上的点，但是也可以用命令 glEdgeFlag*()手工控制这些边界标记的设置。该命令需要在 glBegin()和 glEnd()命令对之间调用。下一条 glEdgeFlag()命令调用之前，它将影响所有的指定顶点。它仅适用于为多边形、三角形和四边形指定顶点，但不适于为三角形切片或四边形切片指定顶点。

void glEdgeFlag(GLboolean flag);
void glEdgeFlagv(const GLboolean *flag);

指出一个顶点是否作为多边形边缘线的初始点。如果 flag 为 GL_TRUE，边缘标识设置为 TRUE（缺省情况），在该函数以 flag 为 0 再次调用之前，创建的任何顶点都被看作在边界线之前。

下面给出一个例子。例 3-3 绘出了图 3-12 所示的轮廓图。

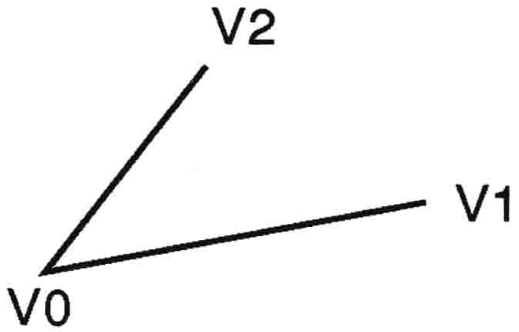


图 3-12　用边界标识绘出的多边形轮廓

例 3-3　多边形边界线的标识

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_POLYGON);
    glEdgeFlag(GL_TRUE);
    glVertex3fv(V0);
    glEdgeFlag(GL_FALSE);
    glVertex3fv(V1);
    glEdgeFlag(GL_TRUE);
    glVertex3fv(V2);
glEnd();
```

3.4　法线向量

法线向量（简称法线）是一个垂直于表面方向的向量。对于一个平坦表面，一个垂向就标识出所有点的法向，但是对于一般曲面，每一点的法线方向都不同。在 OpenGL 中，程序员是可以为每个顶点指定法线的。顶点可以共用一条法线，但是不能在顶点以外的其他地方指定法线。

一个物体的法线向量定义了其表面在空间的方向——特别是其相对于光源的方向。OpenGL 利用这些向量来确定物体在顶点处接收了多少光。光照本身是一个较大的主题，将在第 6 章讨论，在阅读完那一章后，可能需要回顾这里将要介绍的知识。因为程序员一般在定义物体的几何特性的同时为该物体定义法线向量，所以我们在此处先对法线进行简要的讨论。

程序员可以用 glNormal*()来设置传递参数的当前法线。接下来调用 glVertex*()命令指定的顶点赋以当前法线。通常，每个顶点有一个不同的法线，这就需要像下面那样进行一系列的交替调用。

```
glBegin (GL_POLYGON);
   glNormal3fv(n0);
   glVertex3fv(v0);
   glNormal3fv(n1);
   glVertex3fv(v1);
   glNormal3fv(n2);
   glVertex3fv(v2);
   glNormal3fv(n3);
   glVertex3fv(v3);
glEnd();
```

---

void glNormal3{bsidf}(TYPEnx, TYPEny, TYPEnz);
void glNormal3{bsidf}v(const TYPE *v);

按参数的指定对当前法线向量进行设置。函数的非向量形式（没有 v）有三个参数，所以需要指定一个向量（nx, ny, nz）来设置法线。程序员也可以选择使用此函数的向量形式（含有 v)，并提供一个三元素数组来定义期望的法线。该函数的 b, s 和 I 不同形式将其参数值经线性比例变换到[-1.0,1.0]范围内。要为每个物体寻找法线虽然并不是什么不可逾越的难事，但程序员常需要完成一些计算——可能包括求导，才能利用法线技术达到特定的效果。附录 6 解释了如何为任意表面确定法线向量。如果读者对此已经熟稔，或者总可以通过其他途径得到法线，或者干脆不需要用到光照工具，则不需阅读此附录。

注意：在一个表面上的给定点处，有两个方向相反的向量垂直于此表面。习惯上，法线是指向模型表面外侧的向量。（如果在程序员的模型中将内侧和外侧进行了交换，只要将所有的法线向量从（x, y, z）变为（-x, -y, -z）即可）。

另外，还要记住法线向量仅仅表示方向，其长度是无关紧要的。程序员可以将法线指定为任意长度，但是在光照计算执行之前，必须将其变换为长度 1（长度为 1 的向量称为单位长度，或归一化长度）。因此，一般情况下应将法线归一化。若模型变换只包括旋转和平移，这些向量将始终保持其单位长度。变换将在第 4 章中详细讨论。如果执行了不规则的变换（如比例变换或乘以一个剪切矩阵），或者是程序员定义了一个非单位长度法线，就应该在变换之后让 OpenGL 自动地将法线向量归一化。为实现此过程，应以参数 GL_NORMALIZE 调用 glEnable()。在缺省情况下，自动归一化处于失效状态。在 OpenGL 的某些具体实现中，

自动归一化可能需要完成一些额外的计算，从而使程序的性能降低。

3.5　创建表面的多边形模型需注意的问题

下面所提供的一些技术，可能在创建曲面的多边形近似时会用到。在阅读完第 7 章“光照”的内容及第 5 章的“显示列表”的内容后，可能需要回顾一下这部分内容。一旦物体被绘制出之后，光照条件将影响其观察效果，当这些技术与“显示列表”一起应用时，将是十分有效的。在阅读这些技术时，要记住当激活光照计算时，必须定义法线向量才能得到正确的结果。

创建表面的多边形近似是一门艺术，经验的地位是不可替代的。本节中给出的是使入门变得轻松一些的要点。

首先要保证垂直多边形方向一致。这就是要确定从外面观察时，表面上所有的多边形方向相同（都为顺时针或都为逆时针）。这一点一定要在最开始注意，在后来才想起来去弥补将是非常痛苦的。

然后对表面进行分割，注意避免出现非三角形的多边形。三角形的三个顶点可以保证在同一个平面内，而对于四个或更多顶点的多边形，这一点就不能得到保证。从某些角度观看时，非平面多边形的边可能是互相交叉的，OpenGL 无法对这类多边形进行正确的渲染。

在显示速度和图像质量之间总是需要进行折衷考虑的。如果将曲面划分为数量较少的一些多边形，那么渲染的速度会很快，但图像将呈现出参差不齐的现象。如果将其划分为成百万的小多边形，那么图像的质量会非常理想，但渲染的时间将大大增加。程序员最好能够为分割函数提供一个参数，来指定如何按期望效果进行划分，如果物体离眼睛较远，可以划分得粗糙一些。同样，当进行划分时，在表面相对平坦处可以用相对较大的多边形，而在曲率较高的地方——或是高质量的图像，则用小的多边形。还有一个有意义的办法是对轮廓边缘处进行比内部更细致的划分。但是如果表面相对眼睛进行旋转，这样做将很困难，因为轮廓边缘在不停移动。轮廓边缘位于法线向量垂直于表面到视点的向量处——这就是说，当法线和该向量点积为零时，则为轮廓边缘。该点积趋于零时，要求程序员进行更为细致的划分算法。

在模型中一定要设法避免 T 型交叉，如图 3-13 所示。这时无法保证线段 AB 和 BC 与线段 AC 的像素完全重合。究竟能否重合取决于变换和方位。这样会导致在表面上间歇地出现裂缝。

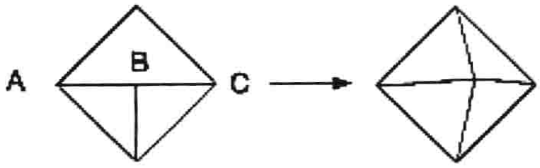


图 3-13　对不期望的 T 型交叉进行修正

如果程序员要创建一个闭合曲面，一定要保证在闭合回路的开始和结束处使用的坐标值完全相同，要不然数值的舍入将导致缝隙和裂纹。下面的例子是一个二维情况下的错误代码。

```
/* 不要使用以下代码 */
    #define PI 3.14159265
    #define EDGES 30
/* 画一个圆 */
    for (i = 0; i < EDGES; i++) {
      glBegin(GL_LINE_STRIP);
        glVertex2f(cos((2*PI*i)/EDGES), sin((2*PI*i)/EDGES);
        glVertex2f(cos((2*PI*(i+1))/EDGES),  sin((2*PI*(i+1))/EDGES);
      glEnd();
    }
```

只有计算机计算出 0 和(2*PI*EDGES/EDGES)的正弦值和余弦值，且两个角度的计算结果相同，边才能正确地相交。当计算机的浮点处理器能够正确做到这一点，就不会有任何问题。但我们仍建议将这段代码进行小小的修改：当 i=EDGES-1 时，用 0 而不要用 2*PI*EDGES/EDGES 来计算正弦值和余弦值。

注意：如果网格化不十分理想，所有的改变将都是可以看见的。在一些动画中，这种变化会带来很大的麻烦。

实例：创建一个二十面体

为了说明在曲面近似时的种种考虑，让我们来看一些例程源代码。这段代码涉及到一个正二十面体的所有顶点（正二十面体是一个理想的实体，由二十个面组成，跨越十二个顶点，每一个面是一个等边三角形）。一个二十面体可以看成是球体的一个粗糙的近似。例 3-4 定义了组成二十面体的顶点和三角形，然后画出该二十面体。

例 3-4　绘制一个二十面体

```
#define X .525731112119133606
#define Z .850650808352039932


static GLfloat vdata[12][3] =
   {{-X, 0.0, Z}, {X, 0.0, Z}, {-X, 0.0, -Z}, {X, 0.0,-Z},
   {0.0, Z, X}, {0.0, Z, -X}, {0.0, -Z, X}, {0.0, -Z, -X},
   {Z, X, 0.0}, {-Z, X, 0.0}, {Z, -X, 0.0}, {-Z, -X, 0.0}
 };
static GLint tindices[20][3] =
   {{0,4,1}, {0,9,4}, {9,5,4}, {4,5,8}, {4,8,1},
```

```
    {8,10,1}, {8,3,10}, {5,3,8}, {5,2,3}, {2,7,3},
    {7,10,3}, {7,6,10}, {7,11,6}, {11,0,6}, {0,1,6},
    {6,1,10}, {9,0,11}, {9,11,2}, {9,2,5}, {7,2,11} };
for (i = 0; i < 20; i++) {
    /* 此处应为颜色信息 */
  glBegin(GL_TRIANGLE);
    glVertex3fv(&vdata[tindices[i][0]][0]);
    glVertex3fv(&vdata[tindices[i][1]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
  glEnd();
}
```

X 和 Z 选择了不规则数据是为了使二十面体的任何一个顶点到原点的距离均为 1.0。十二个顶点的坐标值在数组 vdata[][]中给出，第 0 个顶点为{-X, 0.0, Z}，第一个为{X, 0.0, Z}，依此类推。数组 tindices[][]给出了如何将顶点连接成三角形。例如，第一个三角形由第 0、第 4 和第 1 个顶点构成。如果以给定顺序取顶点，所有的三角形将有同样的方向。

注释为“颜色信息”的程序行可以替换为一条设置第 i 个面的颜色的命令。如果这里没有出现任何代码，所有的面都用同一颜色画出，那样将无法辨别物体的三维特性。另一种明确定义颜色的办法是定义表面的法线向量，然后使用光照，这将在下面描述。

注意：在本节中描述的所有例子中，如果表面并不是一次性画出，那么需要程序员保存计算出来的顶点和法线坐标，以便在绘制表面时不必重复计算。这一点可以通过用用户自己的数据结构或是创建显示列表来实现（见第 5 章）。

1. 定义二十面体的法线

如果对这个二十面体进行光照处理的话，则需要程序员在表面上应用法线向量。在二十面体的每一个平面上，所有的定义一个表面的三个顶点具有相同的法线向量。因此，对任何一组三个顶点，法线只需要指定一次。为画出二十面体，可以用例 3-5 中的代码来代替例 3-4 中“此处为颜色信息”的程序行。

例 3-5　为二十面体指定法线

```
GLfloat d1[3], d2[3], norm[3];
for (j = 0; j < 3; j++)
  d1[j] = vdata[tindices[i][0]][j]
  vdata[tindices[i][1]][j];
  d2[j] = vdata[tindices[i][1]][j]
  vdata[tindices[i][2]][j];
}
normcrossprod(d1, d2, norm);
```

```
glNormal3fv(norm);
```

函数 normcrossprod()生成两个向量的归一化叉积，如例 3-6 所示。

例 3-6　计算两个向量的归一化叉积

```
void normalize(float v[3])
  GLfloat d = sqrt(v[1]*v[1]+v[2]*v[2]+v[3]*v[3]);
  if (d == 0.0) {
    error("zero length vector");
    return;
  }
  v[1] /= d; v[2] /= d; v[3] /= d;
}

void normcrossprod(float v1[3], float v2[3], floa out[3])
{
  GLint i, j;
  GLfloat length;
  out[0] = v1[1]*v2[2] - v1[2]*v2[1];
  out[1] = v1[2]*v2[0] - v1[0]*v2[2];
  out[2] = v1[0]*v2[1] - v1[1]*v2[0];
  normalize(out);
}
```

如果用一个二十面体来近似一个有阴影的球体，就需要用垂直于真实球体表面的，而不是垂直于二十面体表面的法线向量。对于一个球体，法线向量是简单的，即每一个都指向从原点到相应顶点的向量所指的方向。由于二十面体的顶点数据对应着半径为 1 的球体，所以法线和顶点数据是一样的。下面的代码就将绘出一个二十面体对一个平滑的有阴影的球体的近似（假设激活了光照状态，如第 7 章所述）。

```
for (i = 0; i < 20; i++)
  glBegin(GL_POLYGON);
    glNormal3fv(&vdata[tindices[i][0]][0]);
    glVertex3fv(&vdata[tindices[i][0]][0]);
    glNormal3fv(&vdata[tindices[i][1]][0]);
    glVertex3fv(&vdata[tindices[i][1]][0]);
    glNormal3fv(&vdata[tindices[i][2]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
  glEnd();
```

```
}
```

2. 对模型进行改进

一个由二十面体近似的球体只有当屏幕图像较小时才能有较好的效果，但是有一个更简单的办法来提高近似的精度。假设二十面体内接于球体，并如图 3-14 所示对三角形进行再分割。由于新引进的顶点稍稍地处在球内部，通过归一化将这些顶点移到球表面上（除以一个因子，使其长度变为 1）。通过重复这一分割过程可以得到任意的精度的近似。在图 3-14 中显示了三个分别使用了二十、八十和三百二十个近似三角形的球体。

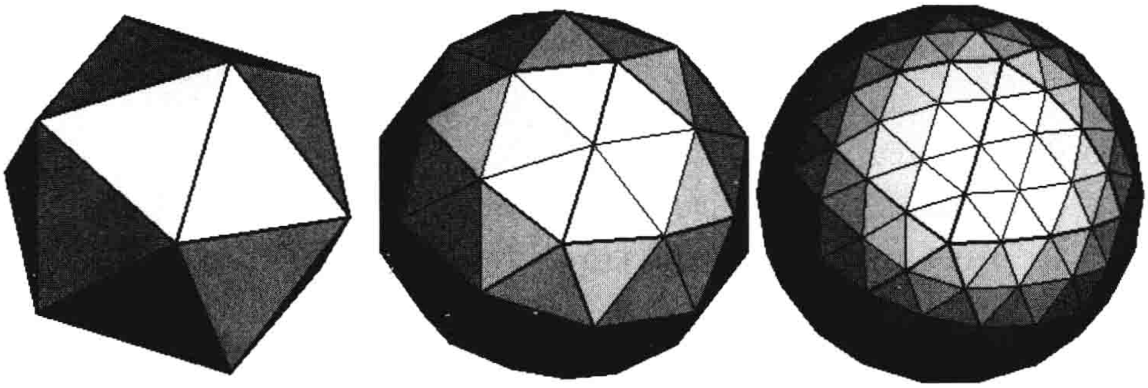


图 3-14　利用分割来提高曲面的多边形近似精度

例 3-7 完成一次单一分割，创建一个八十面的近似球体。

例 3-7　单一分割

```
void drawtriangle(float *v1, float *v2, float *v3)
{
  glBegin(GL_POLYGON);
    glNormal3fv(v1); vlVertex3fv(v1);
    glNormal3fv(v2); vlVertex3fv(v2);
    glNormal3fv(v3); vlVertex3fv(v3);
  glEnd();
}
void subdivide(float *v1, float *v2, float *v3)
{
  GLfloat v12[3], v23[3], v31[3];
  GLint i;
  for (i = 0; i < 3; i++) {
    v12[i] = v1[i]+v2[i];
    v23[i] = v2[i]+v3[i];
    v31[i] = v3[i]+v1[i];
```

```
    }
    normalize(v12);
    normalize(v23);
    normalize(v31);
    drawtriangle(v1, v12, v31);
    drawtriangle(v2, v23, v12);
    drawtriangle(v3, v31, v23);
    drawtriangle(v12, v23, v31);
}
for (i = 0; i < 20; i++) {
    subdivide(&vdata[tindices[i][0]][0],
            &vdata[tindices[i][1]][0],
            &vdata[tindices[i][2]][0]);
}
```

例 3-8 对例 3-7 进行了小小的修正，递归地将三角形分割为恰当的深度。如果深度值为 0，不进行任何分割，三角形是什么样的就怎么画。如果深度值为 1，将执行一个单一分割，以此类推。

例 3-8 递归分割

```
void subdivide(float *v1, float *v2, float *v3, long depth)
{
    GLfloat v12[3], v23[3], v31[3];
    GLint i;
    if (depth == 0) {
        drawtriangle(v1, v2, v3);
        return;
    }
    for (i = 0; i < 3; i++) {
        v12[i] = v1[i]+v2[i];
        v23[i] = v2[i]+v3[i];
        v31[i] = v3[i]+v1[i];
    }
    normalize(v12);
    normalize(v23);
    normalize(v31);
    subdivide(v1, v12, v31, depth-1);
```

```
    subdivide(v2, v23, v12, depth-1);
    subdivide(v3, v31, v23, depth-1);
    subdivide(v12, v23, v31, depth-1);
}
```

3. 通用分割法

例 3-8 中描述的递归分割技术可以用于其他类型的表面。一般的，在某个深度或曲度条件得以满足（精细的分割使表面曲度大的部分呈现出较好的效果）时，递归结束。

为了寻求分割问题的更一般的解决办法，下面考虑以两个变量 u[0]和 u[1]为参数的任意曲面。假定给定了两个子程序：

```
void surf(GLfloat u[2], GLfloat vertex[3], GLfloat  normal[3]);
float curv(GLfloat u[2]);
```

如果函数 surf()传递参数 u[]，返回值为相应的三维顶点和法线向量（长度为 1）。如果将参数 u[]传递给 curv()，返回值为表面上该点的曲度（关于计算表面曲度的详细信息请参见相关的微分几何教程）。

例 3-9 示例了一个递归子程序，该程序对一个三角形进行分割，直到达到了最大深度或者在三个顶点处的最大曲率小于某个限定值为止。

例 3-9　通用分割法

```
void subdivide(float u1[2], float u2[2], float u3[2], float cutoff, long depth)
{
    GLfloat v1[3], v2[3], v3[3], n1[3], n2[3], n3[3];
    GLfloat u12[2], u23[2], u32[2];
    GLint i;
    if (depth == maxdepth || (curv(u1) < cutoff &&
        curv(u2) < cutoff && curv(u3) < cutoff)) {
        surf(u1, v1, n1); surf(u2, v2, n2); surf(u3, v3, n3);
        glBegin(GL_POLYGON);
           glNormal3fv(n1); glVertex3fv(v1);
           glNormal3fv(n2); glVertex3fv(v2);
           glNormal3fv(n3); glVertex3fv(v3);
       glEnd();
return;
    }
    for (i = 0; i < 2; i++) {
        u12[i] = (u1[i] + u2[i])/2.0;
        u23[i] = (u2[i] + u3[i])/2.0;
```

```
        u31[i] = (u3[i] + u1[i])/2.0;
    }
    subdivide(u1, u12, u31, cutoff, depth+1);
    subdivide(u2, u23, u12, cutoff, depth+1);
    subdivide(u3, u31, u23, cutoff, depth+1);
    subdivide(u12, u23, u31, cutoff, depth+1);
}
```

第 4 章　视图

本章主题

在第 3 章中解释了如何通过 OpenGL 绘制出将显示在屏幕视景中的几何模型。现在需要确定的是模型应该怎样放置在视景中，并选择一个观察视景的有利位置。程序员可以使用缺省的定位和缺省的有利观察位置，但大多数情况下可能更希望能够自己指定。

读者一定还记得第 1 章中介绍的彩图中的一系列场景。产生这个图像的程序包含一个构建模块的单一几何描述。视景中每个模块都得以精细的定位：一些模块分散在地板上，一些在桌子上堆积起来，另一些聚集成球体。同时，我们选择了一个特殊的视角，很明显，我们希望看到放置球体的屋子的角落。但是究竟距离视景应该多远呢？观察者应该在什么位置呢？我们要确保最终的视景图像能够包括窗外的景色，能够看到一部分地板，所有的物体不仅要被看到还要进行有趣味的布置。这一章将介绍怎样用 OpenGL 实现这些任务：怎样在三维空间中确定模型的位置和方向，怎样在三维空间中建立视点的位置。上述因素将决定图像在屏幕上的视觉效果。

需要记住的一点是，计算机绘图是产生三维物体的二维图像（因为在屏幕上只能画出二维图像）。但是在屏幕上绘制图形的时候，必须在三维坐标系下来考虑画法。人们很容易犯这样一个错误，在想创建一个三维图形时，人们首先考虑最终的二维平面图像是怎样的。这里提醒读者注意，在应用 OpenGL 来编制程序时，要避免总是琢磨哪些像素需要被画出来，哪些不需要画出来。需要程序员完成的，是想象出一个三维空间，所有的操作都在这个空间中进行。这就相当于假想模型处在隐藏于计算机内部的某个三维空间中，究竟哪些像素点需要被画出来，完全交给计算机来决定。

由下面所列的三种计算机函数组成的一系列操作，可以将一个物体的三维空间坐标变换成屏幕上的像素点位置。

1）变换，由矩阵乘法来表达，包括模型化操作、视图操作和投影操作。这些操作中包括旋转、平移、比例缩放、反射、正交投影以及透视投影。通常，可以利用几种变换的组合来描绘一个场景。

2）由于视景是在矩形窗口中渲染，处于窗口外的物体（或其一部分）必须裁剪掉。在三维计算机图形学中，当物体超出裁剪平面的一边时就执行裁剪操作。

3）最后，必须建立需要变换的坐标与屏幕像素之间的对应关系，这就是所谓的视区变换。

在本章接下来的几节中，将详细描述所有的这些操作，并将讨论怎样控制这些操作。

- 概述：照相机类比法，用照相机拍照做类比，概要地说明了变换过程，同时给出了一个简单的对物体进行变换的例程，并通过该程序简要地描述了基本的 OpenGL 变换函数。
- 视图变换和模型变换，详细解释了怎样指定设想视图和模型变换的效果。这些变换确定了模型与相机的相互关系，以便获得期望的最终图像。
- 投影变换，描述了怎样设定视图体的形状和方向。视图体决定了一个视景如何投射到屏幕上（利用透视投影或正交投影），以及场景之外的哪些物体或物体的哪一部分需要被裁剪下去。
- 视区变换，解释了怎样控制三维模型坐标到屏幕坐标的变换。
- 变换的诊断，给出了一些有用的提示，在程序员无法通过自己的模型变换、视图变换或视区变换达到预期的效果时，能够借以找到原因。
- 操作矩阵堆栈，讨论了怎样保存和恢复某特定变换。这一点在绘制由简单物体构成的复杂物体时是非常有用的。
- 附加裁剪平面，描述了如何指定由视图体定义的那些平面以外的附加裁剪平面。
- 组合变换举例，通过两个较为复杂的应用，让读者在“变换”操作中漫游。

4.1 概述：照相机类比法

4.1.1 照相机类比法及其在计算机中的实现

在计算机绘图中，对场景进行变换的整个过程类似于用一台相机拍摄照片。如图 4-1 所示，相机拍摄照片的具体步骤（括号内表示相应的计算机绘图操作）如下：

1）将相机固定在三角架上并使相机对准场景（视图变换）。

2）使想要拍摄的场景处于取景框中的合适位置（模型变换）。

3）选择相机镜头或调整放大倍数（投影变换）。

4）决定最终照片尺寸的大小，比如想要得到的是一张经过放大后的照片（视区变换）。

以上步骤与程序中指定的变换顺序相对应，然而这一顺序并不一定是对物体顶点所作的相关数学操作的顺序。在源代码中视图变换一定要在模型变换之前执行，但是投影变换和视区变换可以在进行绘图之前的任何时刻执行。图 4-2 显示了在计算机中进行上述操作的顺序。

为了指定视图、模型和投影三个变换，需要构造一个 4 阶方阵 M，然后将此矩阵乘以场景中的顶点坐标 v，完成变换 $v'=Mv$。

顶点具有四个坐标值（x, y, z, w），大多数情况下 w 的值为 1，对二维数据来说，z 值为 0。这里需要注意的是，指定的视图和模型变换将自动地作用在物体的表面法线向量以及各个顶点上（法线向量仅在人眼坐标系中起作用）。这样就确保了法线向量与顶点数据之间的相互关系不会发生改变。

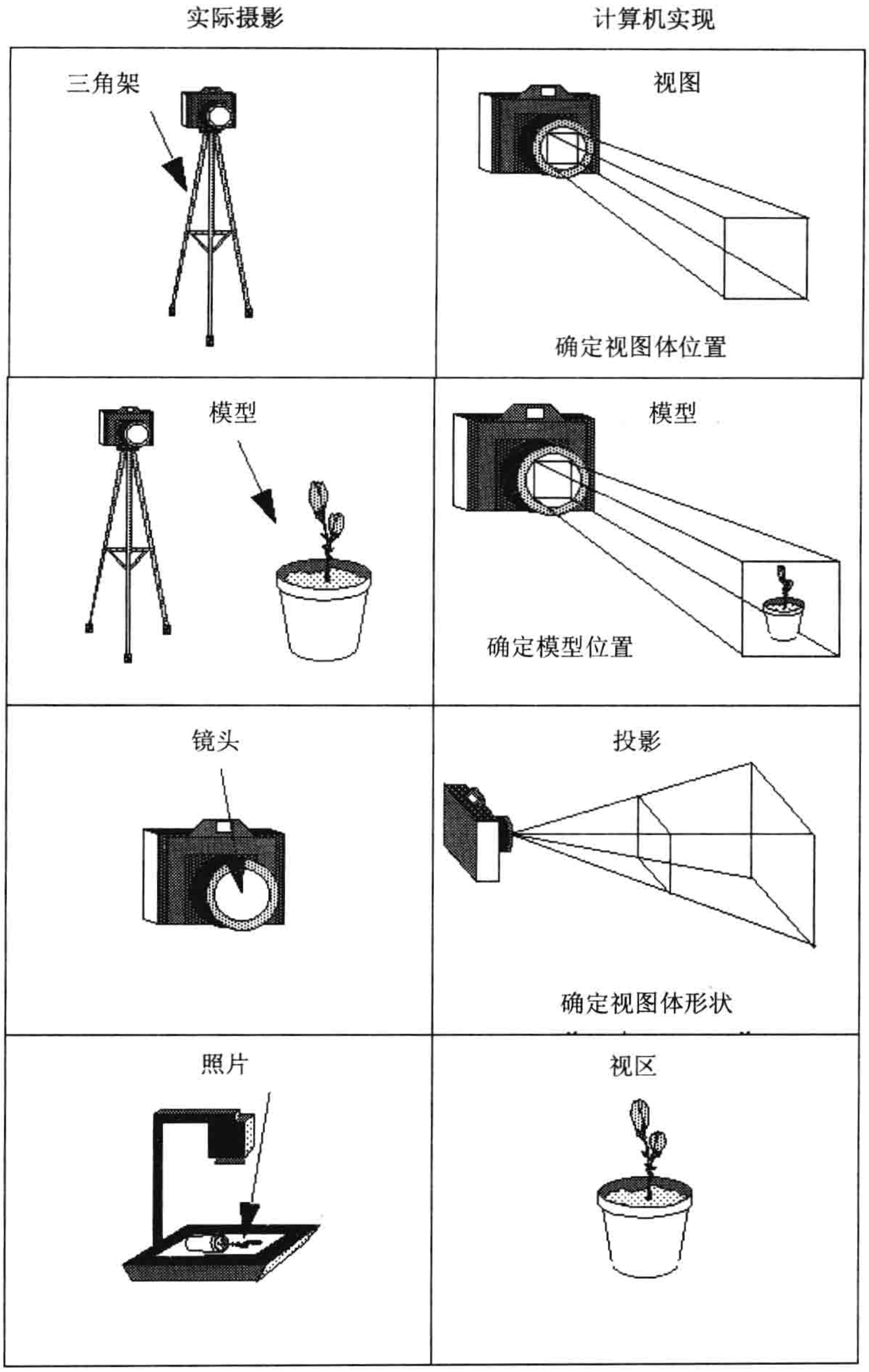


图 4-1　照相机类比法

对指定的视图与模型变换进行组合得到模型视图矩阵，而模型视图矩阵作用于物体坐标后，就得到人眼坐标系下的场景图像。接下来，考虑到视角及观察点等因素的影响，我们要将场景中的某些物体去掉（即不在场景中画出来），而在一些具体的实际应用中，或者要得到物体的截面视图，这时就需要应用到裁剪面的概念。这些裁剪面既可以是在计算机内已经指定的，也可以是程序员任意指定的。

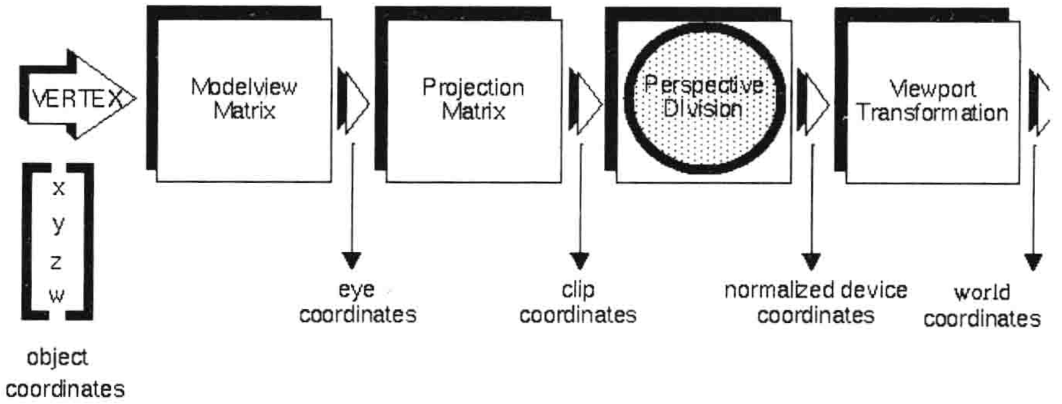


图 4-2　顶点变换的过程

OpenGL 利用投影矩阵产生裁剪坐标。这个变换定义了一个视图体，视图体以外的物体将被裁剪掉，这样在最终的场景中，就可以不画出这些物体。在裁剪操作之后，物体顶点的坐标值除以 w 实现透视分割，以产生归一化的设备坐标（附录 7 提供了关于 w 坐标和它对矩阵变换影响的详细说明）。最后，经前面一系列变换后的坐标，通过视区变换，变换到窗口坐标上来。在这一过程中，程序员可以操作视区尺寸的大小，使最终显示在屏幕上的图像放大、缩小或拉伸。

我们可以做这样合理的假定：即只用 x 和 y 坐标就足以确定哪些像素点要画在屏幕上。然而，以上所有的变换也同样作用于 z 坐标，这样，通过变换的 z 值就可以正确地反映出给定顶点的深度（即离开屏幕的距离）。

深度值的一个作用是避免不必要的绘图过程。例如，假定两个顶点具有相同的 x 和 y 坐标，但 z 坐标不同，OpenGL 可以根据 z 值决定哪些物体表面是不可见的，即被其他物体挡住了视线，这样就不必去画那些隐藏在物体后面的物体表面（关于该技术的详细介绍，请参见第 11 章。这一技术被称之为“隐藏表面的消除”）。

现在读者可能已经发现，为了更好地理解本章内容，需要了解一些关于矩阵的数学背景知识。如果想对 OpenGL 的知识了解得更为深入，我们建议读者阅读一些有关矩阵概念及其运算方面的书籍，例如线性代数等等。

4.1.2　一个简单的例子：绘制立方体

例 4-1 绘制一个经模型变换后进行比例缩放的立方体（如图 4-3 所示）。采用的视图变换是沿 z 轴简单地向下平移，同时还指定了投影变换和视区变换的函数形式。本节后面的部分将通过例 4-1 让读者体会这些变换函数的用法，并简要地解释了这些函数的参数和功能。后面各节将对所有的 OpenGL 变换函数加以全面详细的讨论。

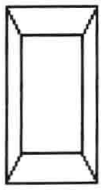


图 4-3　绘制立方体

例 4-1　绘制立方体: cube.c

```
#include <windows.h>
```

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
void CALLBACK display (void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f (0.0, 0.0, 0.0);
   glLoadIdentity ();                  /* 将当前矩阵清为单位阵 */
   glTranslatef (0.0, 0.0, -4.0);         /* 视图变换 */
   glScalef (1.0, 2.0, 1.0);              /* 模型变换 */
   auxWireCube(1.0);                      /* 绘制立方体 */
   glFlush();
}
void myinit (void)
{
   glClearColor(1.0, 1.0, 1.0, 1.0);
    glShadeModel (GL_FLAT);
}
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();                     /* 定义投影变换 */
   glFrustum (-1.0, 1.0, -1.0, 1.0,1.5, 20.0);
   glMatrixMode (GL_MODELVIEW);           /* 返回模型视图矩阵 */
   glViewport (0, 0, w, h);               /* 定义视区 */
}
int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
   auxInitPosition (0, 0, 250, 250);
   auxInitWindow ("编程指南例程 4-1");
   myinit ();
   auxReshapeFunc (myReshape);
   auxMainLoop(display);
    return 0;
```

}

下面对该程序进行剖析。

1. 视图变换

如前所述，视图变换类似于相机的定位和调焦对准。在本例的代码中，在指定视图变换之前，函数 glLoadIdentity()先将当前矩阵设定为单位矩阵。这一步是必不可少的，原因在于，大多数变换函数将指定的矩阵乘以当前矩阵，并将结果放入当前矩阵中。在每一次变换之前，如果程序员没有用单位矩阵替换当前矩阵，则该变换的结果中将带入上一个变换矩阵的影响。虽然在某些特殊情况下可能正是需要通过上一个变换矩阵来实现组合变换，但在一般情况下都必须用单位矩阵替换当前矩阵。

矩阵的初始化一旦完成，就可以由函数 glTranslatef()指定视图变换。该函数体的各个参数分别表示在 x，y 和 z 方向调整（移动）相机的位置。本例中设定的参数值使相机沿 z 轴的负方向移动 5 个单位长度，而 x 和 y 方向上保持不变。在缺省情况下，相机和场景中的其他物体一样，其初始位置在原点处；相机的初始方向为 z 轴的负方向。此处所用的特定视图变换具有使相机离开立方体所在位置的效果，但相机仍然对准物体。如果想改变相机所指的方向，可以调用函数 glRotatef()。视图变换的具体内容将在视图与模型变换一节中详细讨论。

2. 模型变换

利用模型变换确定模型的位置和方向。例如，可以对模型进行旋转、平移和缩放，或者这几种操作的组合。函数 glRotatef()和 glTranslatef()实现模型相对于原点的旋转和平移，这在前面已经提到过。这里，模型变换是通过调用函数 glScalef()来实现的。该函数的参数值表明沿三个坐标轴方向上如何缩放。如果所有的参数均为 1.0，即缩放比率为 1.0，该函数对生成的场景没有影响。在本例中，该函数第二个参数值（y 方向）为 2.0，其结果是立方体在 y 方向上放大了两倍。也就是说，如果该立方体的一个顶点的初始位置在（3.0，3.0，3.0），该顶点经变换后将被画到位置（3.0，6.0，3.0）上，模型变换的结果使立方体发生变形。从图 4-3 中看到的图像已不再是一个立方体，而是一个长方体。

模型变换与视图变换的区别在于视图变换是通过移动相机来实现的，而模型变换是通过改变模型的位置和方向来实现的。视图变换和模型变换在本质上具有的二义性使我们必须同时考虑这两种变换作用的效果，而将这种相互影响分开来考虑是不合理的。不过有时仅仅考虑其中一种方式能够使问题得到简化。

因为视图变换和模型变换是相互影响的，所以在变换之前，要将模型变换和视图变换进行组合运算，得到模型视图矩阵。在视图和模型变换这一节中将详细论述如何考虑模型变换和视图变换，以及如何设定这两种变换以得到期望的结果。

还有一点需要提请读者注意的是，模型变换、视图变换以及绘制立方体的函数 auxWireCube()等都包含在子函数 display()中。这样一来，如果对窗口进行了某些操作（如移动该窗口或者是关闭或挪动覆盖在该窗口上的其他窗口），该进程自动地重新调用函数 display()绘制窗口内的内容，并且可以保证每一次重画时，该立方体都经适当的变换并按期

望的方式画出。反复调用子函数 display()的另一个潜在含义就是强调在执行视图变换和模型变换之前，尤其是在两次 display()调用之间插入了其他变换的时候，必须加载单位矩阵。

3. 投影变换

投影变换类似于选择相机的镜头或者说调整相机的焦矩。可以这样认为，投影变换确定视野的大小或视图体的大小，即确定哪些物体在视野之内，以及它们的形状和大小。这一点相当于选择不同种类的镜头（包括广角镜头、标准镜头及远距照相镜头）以得到不同的效果。例如，用广角镜头可以得到比标准镜头更广的场景，而远距照相镜头可以使拍摄出来的物体看起来比实际距离更近。在计算机图形学中，用不着花一大笔钱去买一个 2000mm 的远距照相镜头，图形计算机能够为程序员提供各种形式的采景方法和假想镜头，用来虚拟各种图像的显示效果。

除了确定视野的大小，投影变换还将决定物体在计算机屏幕上的投影方式。OpenGL 提供了两种投影方式，并且定义了相应的函数，以及描述不同投影方式的相关参数。一种方式是透视投影，类似于在日常生活中观察事物。透视使得越远的物体显得越小。例如，利用透视法进行观察，两条铁轨在很远处交汇成一点。要制作逼真的画面，就必须用到透视投影。在本例中，透视投影是通过调用函数 glFrustum()来实现的。

另外一种投影方式是正交投影，这种方式将物体直接映射到屏幕上，而不改变它们的相对尺寸。正交投影常用在建筑学或计算机辅助设计中，不管视觉效果如何，在这些应用环境下，最终的图像必须准确地反映物体的尺寸。建筑师有时也要使用透视图，但那仅仅是为了显示从各个不同的视点观察时，特定的建筑物和某些内部空间的结构。正交投影图常用于建筑中的设计蓝图和高度图中。在投影变换一节中将更为详细地讨论如何实现这两种类型的投影变换。

在调用函数 glFrustum()设置投影变换之前，还必须做一些准备工作。如本例中的 myReshape()子程序所示，首先调用函数 glMatrixMode()，以 GL_PROJECTION 为参数指定当前矩阵为投影矩阵，此后的变换操作将在此投影矩阵的基础上进行。然后再一次调用函数 glMatrixMode()，其参数为 GL_MODELVIEW，即指定当前矩阵为模型视图矩阵，接下来的变换操作将改变模型视图矩阵，而不再是投影矩阵。关于投影矩阵及模型视图矩阵的详细信息，请参见矩阵堆栈的操作这一节内容。

函数 glLoadIdentity()用于初始化当前投影矩阵，以保证指定的投影变换生效。

接下来调用函数 glFrustum()，其参数定义了投影变换的各种参量。在本例中，投影变换及视区变换都包含在子函数 myReshape()中，当创建、移动窗口或改变窗口形状时，都自动调用该函数。由于投影变换和视景都与屏幕直接相关，尤其与屏幕上的窗口尺寸直接相关，所以，调用子函数 myReshape()是非常有意义的。

4. 视区变换

投影变换和视区变换决定视景映射到计算机屏幕上的方式。 投影变换给定映射的机制，而视区变换则给出映射后图像在屏幕中最终显示的形状。由于视区变换指定了图像在计算机

屏幕中所占的区域，所以，换句话说，视区变换定义了图像最终的大小和位置（比如说是否需要放大或缩小）。

函数 glViewport()的参数给出 用于绘制场景的屏幕区域在窗口中的原点——本例中为（0，0）——以及屏幕区域的高度和宽度，以屏幕上的像素点为单位。如果窗口的大小发生改变，则视区也要相应改变。通常，屏幕区域的宽度和高度是由窗口自身的宽度和高度指定，而不是绝对的尺寸大小，这样一来，视区的大小就能够随着窗口大小的改变而改变。关于如何定义视区的详细信息请参见视区变换一节。

5. 绘制场景

在确定所期望的变换之后，便可以绘制场景（也就是“照相”）。首先，OpenGL 对场景中每一个物体的顶点逐个进行模型变换和视图变换；然后，这些顶点按照给定的投影变换进行变换，并且对落在投影变换所描述的视图体之外（即视景之外）的顶点进行裁剪；最后，其他（指未被裁剪的）变换过的顶点被 w 除，并映射到视区内。

4.1.3 多种用途的变换函数

在这部分内容里将继续讨论 OpenGL 某些函数的功能及其用法，而这些函数对于实现为绘制场景所希望得到的变换是非常有用的。在上面的内容中已经给出了两个这类函数：glMatrixMode() 和 glLoadIdentity()。这里给出另外两个函数：glLoadMatrix*() 和 glMultMatrix*()。通过这些函数，程序员可以直接定义任何一个变换矩阵，然后用此矩阵与当前矩阵作乘法运算。对变换函数更进一步的讨论，如 glTranslate*() 和 glScale*()等函数的用法，将在以后的章节详细论述。

下面就对这两个函数的功能和用法作一些较为详细的说明。

如同在前面的章节中所叙述的那样，在进行变换之前，程序员必须声明修改的是模型视图矩阵还是投影矩阵。这一过程由函数 glMatrixMode()来实现。当使用嵌套的、并且可能进行反复调用的一批 OpenGL 函数时，需要重新正确地设置矩阵模式。函数 glMatrixMode()也可以用于指定一个纹理矩阵。有关纹理操作的详细内容请参见第 10 章。

---

void glMatrixMode(GLenum mode);

对于函数 glMatrixMode()，其参数 Mode 有 GL_MODELVIEW、GL_PROJECTION 和 GL_TEXTURE 三个取值，分别指定模型视图矩阵、投影矩阵和纹理矩阵等三个单位矩阵为当前矩阵，其后的操作将在上述某个指定的矩阵基础之上进行变换。参数 Mode 的缺省值为 GL_MODELVIEW，即模型视图矩阵。

考虑到函数 glMatrixMode()之后的变换操作将修改当前矩阵，程序员可以调用函数 glLoadIdentity()将改变后的当前矩阵用相应的单位矩阵替换，以恢复初始的状态。一般地，在给定投影或视图变换之前常常需要调用这一函数，而在给定模型变换之前也可以调用这一

函数。

---

void glLoadIdentity(void);

函数 glLoadIdentity()指定一个 4 阶单位矩阵为当前矩阵，该单位矩阵对应于函数 glMatrixMode()所设定的矩阵模式（模型视图矩阵、投影矩阵或纹理矩阵）。类似于 glLoadIdentity()的函数还有 glLoadMatrix*()和 glMultMatrix*()。函数 glLoadMatrix*()载入某个特定矩阵作为当前矩阵，而函数 glMultMatrix*()将其参数（一个矩阵）与当前矩阵相乘，并将结果返回给当前矩阵，也就是说，运算后的当前矩阵等于运算前的当前矩阵与函数的矩阵参数相乘。这两个函数的参数都是一个包括 16 个变量值的向量（m1, m2, ... , m16)，记为矩阵 M。

为了提高程序运行的效率，可以使用显示列表存储经常用到的矩阵（和它们的转置——在 OpenGL 程序的运行过程中，常常需要计算模型矩阵的转置，以使法线和裁剪平面可以正确的变换到人眼坐标中。)，而不是在程序中循环地做一些重复的矩阵运算。(参见显示列表设计准则。)

需要注意的是，用 C 语言实现 OpenGL 的操作时，声明一个 m[4][4]的矩阵，则元素 m[i][j]代表 OpenGL 转移矩阵的第 i 列第 j 行的变量值，这与标准 C 的惯例（m[i][j]为第 i 行第 j 列的元素）恰好相反。因此，为了避免出错，建议最好将自己的矩阵定义为 m[16]的形式。

---

void glLoadMatrix{fd}(const TYPE *m);

这里，指针 m 是一个 4 × 4 的矩阵，或者说是一个长度为 16 的数组。指定该矩阵为当前矩阵。

---

void glMultMatrix{fd}(const TYPE *m);

同样，指针 m 是一个 4 × 4 的矩阵，该矩阵与当前矩阵相乘，并将结果返回给当前矩阵。

OpenGL 的矩阵乘法：假定当前矩阵为 C，由函数 glMultMatrix*()或其他变换函数给定的矩阵为 M，则矩阵相乘的结果为 CM。由于矩阵乘法的不可交换性，即两个矩阵相乘的顺序不同，其结果也不同，因此，在使用 glMultMatrix*()等函数时需要明确矩阵相乘的顺序，从而正确地设定参数值，以得到期望的变换效果。

4.2　视图变换和模型变换

正如 4.1.2“一个简单的例子：绘制立方体”一节中所提及的那样，在 OpenGL 中视图

变换和模型变换是相互关联的，实际上通过运算结合为一个模型视图矩阵。对于一个计算机图形学的初学者来说，其面对的最困难的问题就是理解组合三维变换的作用。如同读者已经看到的那样，可以在两种办法中任选其一来执行变换操作——或者在一个方向上移动照相机，或者向相反的方向移动物体。每一种办法都有利有弊，但在一些情况下其中一种方法可能与所期望的变换匹配得更自然。如果能够为特定应用程序找到一个比较自然的近似，这将使必要的变换变得简单而形象，并易于写下相应的代码来指定对矩阵的操作。在这一节中，首先讨论不同变换操作的效果，然后，给出用于变换操作的函数。在本节，仅仅使用了读者在本书中已经见过的矩阵操作函数。这里，需要强调的是，在执行模型变换或视图变换之前，程序员必须使用参数 GL_MODELVIEW 调用函数 glMatrixMode()。

4.2.1 关于变换的说明

这里以一个简单的例子开始我们的讨论。在这个例子中包括两个变换：以原点为圆心，绕 Z 轴逆时针 45 度的旋转和沿 X 轴的平移。首先，假定物体的大小比物体平移的距离小得多，这样一来可以明显地看到经过变换后的效果，并且物体的最初位置位于原点。如果将物体先进行旋转然后平移，变换后的物体出现在 X 轴上。但是若先沿 X 轴平移然后绕原点旋转，物体出现在直线 y=x 上。上述两种结果如图 4-4 所示。从这个例子中我们可以看出，OpenGL 绘图对变换顺序的要求是十分严格的。先做变换 A 然后做变换 B，肯定得到与先 B 后 A 截然不同的结果。

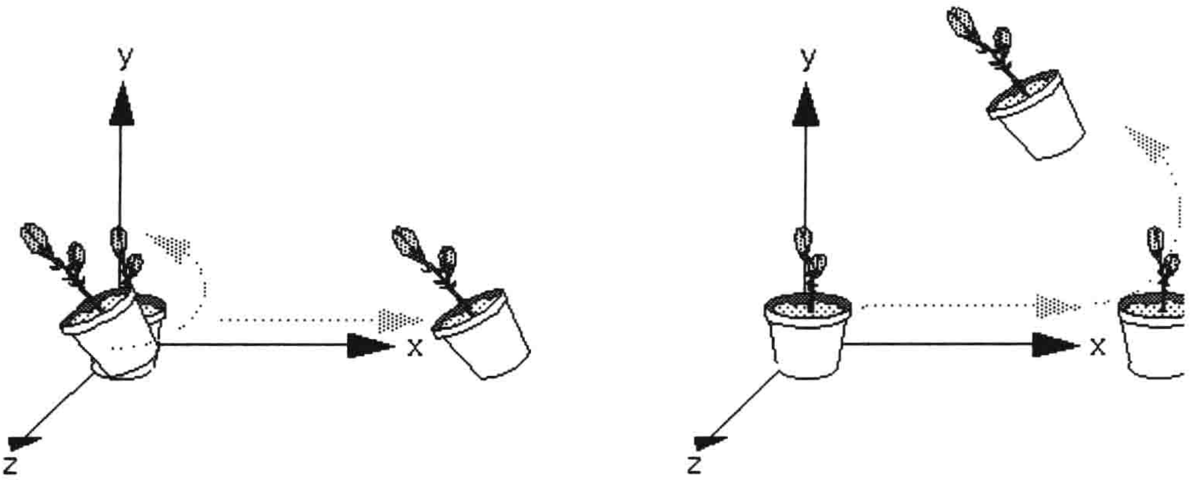


图 4-4 先旋转再平移与先平移再旋转的不同结果

下面我们讨论如何确定一系列变换操作的顺序。这里，所有的视图变换和模型变换都由 4×4 矩阵表示。每一个 glMultMatrix*()函数或其他变换函数都是在上一个变换得到的当前矩阵 C（模型视图矩阵）的基础上乘以一个该变换函数指定的 4×4 矩阵 M，得到一个新的当前矩阵，值为 CM（即 C=CM）。如此循环，在所有变换操作之后，将物体顶点 v 乘以当前矩阵，得到经过变换后物体图像的最终结果。这一过程意味着在程序中调用的最后一个变

换函数 CMv 实际上才是第一个应用于顶点的函数。因此，为了得到期望的变换，在程序中必须以与实际变换过程相反的顺序进行矩阵运算。举个例子来说明。

在下面的代码序列中，将一个顶点做三次变换：

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(N);          /* 应用变换 N */
glMultMatrixf(M);          /* 应用变换 M */
glMultMatrixf(L);          /* 应用变换 L */
glBegin(GL_POINTS);
glVertex3f(v);             /* 绘制顶点 v */
glEnd();
```

在上述代码中，当前矩阵先后是 I（单位矩阵）、N、NM，最后是 NML，而经过变换后的顶点是 NMLv。这里，期望的顶点变换过程应该为 N(M(Lv))：首先 v 左乘 L，然后 Lv 左乘 M，然后 MLv 左乘 N。从中我们可以看出，期望得到的顶点 v 的变换顺序与程序中定义变换函数的顺序正好相反。事实上，在程序里顶点坐标与当前矩阵只做了一次乘法运算，也就是说，实际的变换过程是 N、M、L 相乘结合成新的当前矩阵 C=NML，然后顶点 v 与当前矩阵 C 相乘 C v，最终得到期望的变换结果。

因此，如果在全局坐标系（或称为固定坐标系，在该坐标系下的矩阵相乘影响到模型的位置、方向和变换比例）中来看，必须使得体现在代码中的变换函数的先后顺序与期望的变换顺序相反。我们再回顾一下图 4-4 中所示的简单例子（绕原点的旋转和沿 X 轴的平移），如果想让物体在变换操作后出现在轴上，就必须先进行旋转，然后进行平移。实现上述变换过程的程序代码如下（R 指旋转矩阵，T 指平移矩阵）：

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(T);                    /* 平移 */
glMultMatrixf(R);                    /* 旋转 */
draw_the_object();
```

还有另一种办法来处理矩阵相乘中的顺序问题。该方法并不考虑模型变换所在的全局坐标系，相反，可以假想一个与所画物体捆绑在一起的局部坐标系，所有的变换操作都相对于这一不断变化的坐标系进行。通过这种途径，代码中矩阵乘法的顺序就是自然的顺序。

下面看看局部坐标在“旋转——平移”例子中是如何实现的。首先想象物体处在与之捆绑在一起的局部坐标系内，平移操作将沿 X 轴同时移动物体和其局部坐标系，然后旋转操作使得物体和局部坐标系同时绕平移后的局部坐标原点旋转，其结果是变换后的物体坐标仍然在局部坐标系的 X 轴上。

这种采用局部坐标系的方法十分适合应用于肩、肘和腕以及每个手指处都有关节的机器

人手臂上。为了描述出手指尖相对于身体的位置，必须从肩开始，向下到腕，并以此类推，在每一个关节处应用恰当的旋转和平移。对于这样一种比较复杂、需要多次相对变换的情况，如果采用全局坐标系的方式进行变换将会使问题变得极其复杂。

然而，如果存在缩放变换，尤其是当缩放变换是非归一化（指在每一个轴上采用不同的比例进行变换）的时候，采取这种局部坐标有可能会产生一些问题。对于归一化的缩放变换，坐标系上的单位长度间距发生改变，则变换也按相应比例移动顶点；而对于非归一化的缩放变换，如果存在旋转变换，局部坐标系的轴与轴之间将不再互相垂直。

由于全局坐标系下矩阵相乘是按相反的顺序进行运算的，所以，视图变换通常发生在模型变换之前。当然，进行视图变换和模型变换是为了得到期望的结果，如果缺省的状态就已经满足要求，那么就没有必要作视图变换或模型变换，这使得问题更加简单。如果没有视图变换，“照相机”将处在缺省位置——原点上，方向为 Z 轴负向；而如果没有模型变换，模型并不移动，它将保持给定的初始位置、方向和尺寸。

一般地，程序员都是先写下所有组成场景的必要的代码，包括变换到相对其他物体正确的位置和方向上，然后再决定视点应该处在场景中的哪个相对位置上，并据此编写视图变换的代码。由于执行模型变换的函数也可以用来执行视图变换，同时先模型变换后视图变换的顺序与许多程序员编制程序时的思路是一致的，所以，尽管在实际变换操作中先进行视图变换，在这里我们还是先讨论模型变换。

4.2.2 模型变换

OpenGL 中进行模型变换的三个子函数分别是：glTranslate*()，glRotate*()和 glScale*()。不难想象，这些子函数将通过移动、拉伸或缩减来变换一个物体（有时是一个坐标系）。这些函数的作用相当于先产生一个恰当的平移矩阵、转动矩阵或是比例变换矩阵，然后以该矩阵为参数调用函数 glMultMatrix*()。当然，直接应用这些函数要比用函数 glMultMatrix*()的速度快，而且 OpenGL 自动为程序员计算这些矩阵。如果读者对细节感兴趣的话，请参见附录 7。

下面对变换函数作一个总结。需要声明的是，这里采用全局坐标系，矩阵运算的操作或者是针对几何物体的顶点，或者是针对附在物体上的局部坐标。

1. 平移

---

void glTranslate{fd}(TYPEx, TYPE y, TYPEz);

参数 x、y、z 定义一个矩阵，当前矩阵乘以该矩阵，其结果是物体在三个坐标轴上发生平移，平移距离的大小为 x、y、z（或以相同的量移动局部坐标系），如图 4-5 所示。

如果函数 glTranslatef*()的三个参数值都为 0.0，称为单位操作，该变换操作对物体或其局部坐标没有影响。

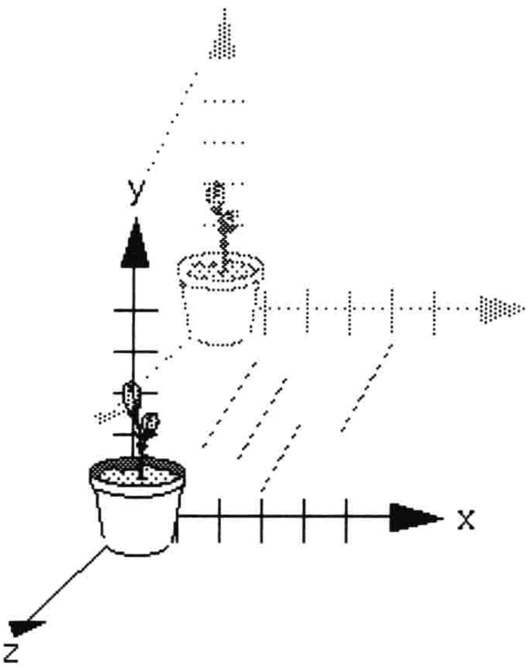


图 4-5　平移物体

2. 旋转

---

void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);

当前矩阵乘以一个矩阵，此矩阵使一个物体（或局部坐标系）绕从原点到点（x, y, z）的线，沿逆时针方向旋转。参数 angle 给出旋转的角度值（单位为角度）。

函数 glRotatef(45.0, 0.0, 0.0, 1.0)使物体绕 Z 轴旋转 45 度，其变换操作后的效果如图 4-6 所示。

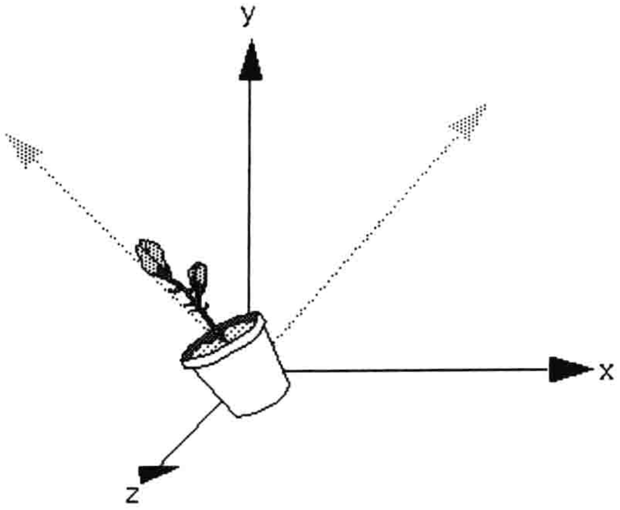


图 4-6　旋转物体

由于物体是绕函数 glRotatef()指定的直线轴（称为旋转轴）旋转，所以距离旋转轴较远的物体比离旋转轴较近的物体旋转得剧烈（有较大的旋转半径）。同样，如果角度参数 angle 为 0，glRotate*()函数将没有任何作用效果。

3. 缩放

---

void glScale{fd}(TYPEx, TYPE y, TYPEz);

参数 x、y、z（分别对应于 x、y、z 坐标）定义一个矩阵，与当前矩阵相乘使得物体沿坐标轴拉伸、缩短以及相对于坐标轴做镜面反射。直观地来看，那就是物体上的每一个顶点坐标 x, y, z 都乘以函数体内相应的 x, y, z 参数。如果采用了局部坐标系，该局部坐标系的各个自由度也乘以 x, y, z 参数，与坐标系内的物体一起缩放。图 4-7 给出了函数 glScalef(2.0, -0.5, 1.0)的作用效果。

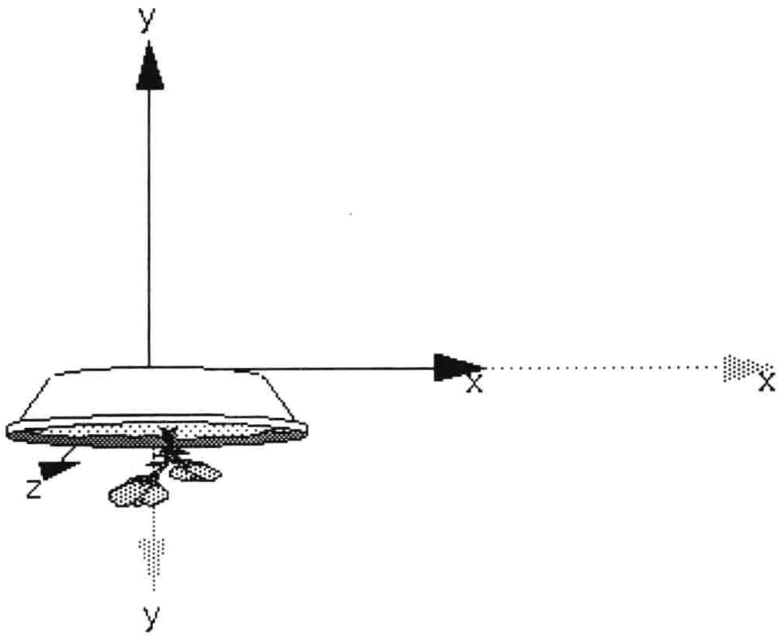


图 4-7 物体的缩放

函数 glScale*()是三个模型变换函数中惟一能改变物体表观尺寸的函数。参数值大于 1.0 时在该方向上拉伸物体，参数值小于 1.0 时在该方向上缩小物体，而参数值为-1.0 时，物体相对于该坐标轴做镜面反射。考虑到变换之间的相互影响，通常应该尽量避免使用函数 glScale*()进行变换，除非在不得已的情况下，同时执行 glScale*()时，由于变换后法线向量必须进行再归一化，故而降低了光照计算的性能。

如果某个参数值为零，则该参数对应坐标轴上的所有物体的坐标都变为零。对于这种情况，需要特别注意，原因在于，从数学角度来讲，此时的矩阵不能被转置，而在某些光照操作中（见第 7 章）必须要用到转置矩阵。当然，在某些特殊情况下，这样做确实是有意义的，

计算物体在平坦表面上的阴影（见阴影）就是一个典型的例子。如果一个坐标系上的坐标要进行衰减为零的操作时，一般用投影矩阵而不是模型视图矩阵。

4. 模型变换的一个例子

例 4-2 是对一个三角形进行四次变换的程序的一部分，其结果如图 4-8 所示，进行的操作如下：

1）绘制一个没有进行模型变换的实体的三角形线框。

2）加入虚线点划和平移后，绘制出这个三角形。

3）由一条点划的长虚线绘制三角形，其高度（Y 轴方向上）减半，而宽度（X 方向上）加倍。

4）画出一个旋转的、经过缩放变换的、由点线组成的三角形。

图 4-8　模型变换的例子

例 4-2　使用模型变换：model.c

```
glLoadIdentity();
glColor3f(1.0, 1.0, 1.0);
draw_triangle();                    /* 实线 */
glEnable(GL_LINE_STIPPLE);          /* 虚线 */
glLineStipple(1, 0xF0F0);
glLoadIdentity();
glTranslatef(-20.0, 0.0, 0.0);
draw_triangle();
glLineStipple(1, 0xF00F);           /* 长虚线 */
glLoadIdentity();
glScalef(1.5, 0.5, 1.0);
draw_triangle();
glLineStipple(1, 0x8888);           /* 点线 */
glLoadIdentity();
glRotatef (90.0, 0.0, 0.0, 1.0);
draw_triangle ();
glDisable (GL_LINE_STIPPLE);
```

这里调用函数 glLoadIdentity()的目的就是消除上一次模型变换的影响，也就是说，通过函数 glLoadIdentity()初始化矩阵值以避免接连的变换产生累积效应。虽然反复使用函数 glLoadIdentity()可以达到期望的效果，但这样做的效率是很低的，而且依赖于特定的 OpenGL 执行工具，更好的解决办法请参见矩阵堆栈这一章节的内容。

有些时候，为了得到物体连续旋转的效果，程序员常常试图通过反复地应用一个值较小的旋转矩阵来实现。采用这一种技术存在的问题是，由于溢出的错误，使得连续的小角度旋转变换导致与期望不相符的结果（甚至不是旋转操作的效果）。因此，应该在每一次更新时增加角度，同时发出一个新的旋转函数。

4.2.3 视图变换

视图变换操作改变视点的位置和方向。如果与照相机拍照相类比，视图变换就起着固定照相机三脚架的作用。使用照相机拍照的时候，需要将照相机平移到某个位置并旋转调整到某个方向，与此相似，视图变换通常也包括平移和旋转两个过程。当然，要在最终的图像或照片中获得某一场景，既可以通过移动照相机，也可以通过将所有的物体对象做反向移动来实现。比如说，在模型变换中将对象逆时针旋转与在视图变换中将照相机顺时针旋转是等效的。但是需要注意的是，必须在执行模型变换函数之前调用视图变换函数，以保证场景中的物体先做模型变换，然后做视图变换。

除了可以选择使用视点的缺省位置和方向，即位于原点、沿 z 轴的负向外，利用视图变换来改变视点，还有如下几种方法：

1）使用一个或多个模型变换函数（即 glTranslate*()和 glRotate*()）。变换的效果既可以看成照相机的移动，也可以看成物体在场景中相对于静止的照相机的移动。

2）使用 OpenGL 实用子程序库的函数 gluLookAt()定义一条视线。该程序封装了一系列的旋转和平移函数。

3）创建用户自己的实用子程序库，在子程序中封装各种旋转和平移函数。在一些特殊的应用中，这样做将方便程序员进行一些特定的视图变换。例如，当飞机飞行时，指定横滚、纵摇和首摇的旋转角度；当照相机沿固定轨道绕物体旋转时，指定相应的极坐标变换等等。

1. 函数 glTranslate*()和 glRotate*()

使用模型变换函数做视图变换，也就相当于将视点以某种预定方式移动同时保持场景中的物体静止。由于视点的初始坐标位于原点，同时习惯上为了简单起见也在原点处构造物体（如图 4-9 所示），这样一来初始的视点就和物体的初始坐标相重合。所以要观察到物体必须进行模型变换或者视图变换操作。

先考虑最简单的情况：将视点后移，以远离物体，这与将物体前移，远离视点是等效的。对于视点来说，缺省的前向是 z 轴的负方向；如果把视点进行旋转，前向的意义则发生变化。例如，通过移动视点将视点与物体拉开 5 个单位，如图 4-10 所示，此时使用的变换函数是：glTranslatef(0.0, 0.0, -5.0)。

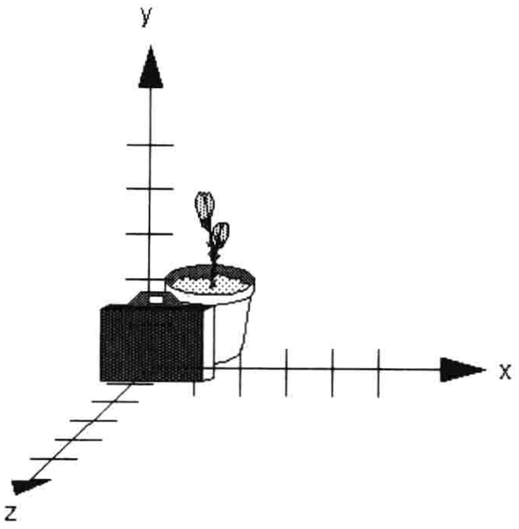


图 4-9 物体和视点同在原点

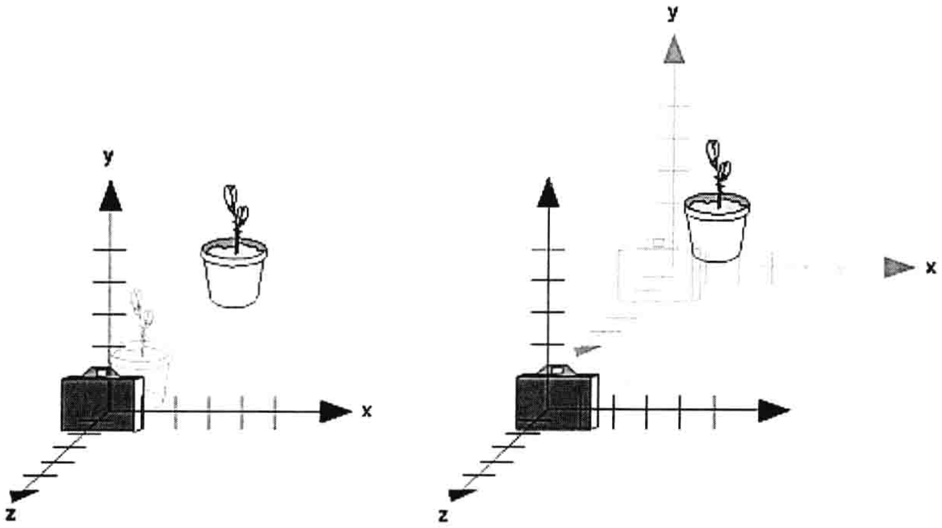


图 4-10 视点和物体的分离

如果希望从侧面观察物体，则需要用到两个变换：旋转和平移。这时候要考虑如何安排这两个变换的执行顺序。以固定的全局坐标系来考虑，首先设想物体和相机都在原点的情况。应该先旋转物体，然后移动物体以远离相机，这样就能够从侧面观察到该物体。按照这种在固定坐标系中的实现方法，变换函数发出的顺序必须与函数作用的顺序相反，因此在程序代码中，需要先编写平移函数然后是旋转函数。

现在考虑采用局部坐标系的实现方法。在这种情况下，首先平移物体及其局部坐标；然后，在当前经过平移的局部坐标系中执行旋转操作。按照这种方法，变换函数就按照其执行的顺序来发送，因此这里先执行的也是平移函数。这样，要得到预想效果，变换函数的先后顺序如下所示：

```
glTranslatef(0.0, 0.0, -5.0);
glRotatef(90.0, 0.0, 1.0, 0.0);
```

如果无法顺利追踪连续的矩阵相乘的作用效果，可以将固定坐标系和局部坐标系两种方法都试用一下，看看哪一种方法更易于理解。有一点需要明确的是，在固定坐标系当中，物体围绕固定坐标系的原点旋转，而在局部坐标系当中，物体围绕局部坐标系的原点旋转。

下面一部分内容将描述采用例程 gluLookAt()的实现方法。

2. 函数 gluLookAt()

通常，程序员在适当的位置构建场景，并希望可以从任意的角度进行观察。函数 gluLookAt()正好具备这样的功能。该函数体一共包括三组参数，分别指定视点的位置，定义一个相机指向的参考点，以及指示哪一个方向向上。首先要选择一个能得到预想场景的视点，而参考点一般位于视景的中间位置。如果已经将场景构建在了原点处，那么参考点通常就是原点。如何确定向上的方向向量要稍微复杂些。假设构造的真实世界场景是在原点，而且 y 轴的正方向向上，那么通常它就作为 gluLookAt()的方向向量（up-vector）。但是，如果是在设计一个飞行仿真器，那么该方向向量就应该始终垂直于机翼向上——当飞机正常飞行时，则“向上”指朝向天空的方向。

函数 gluLookAt()在扫描全景时尤其有用。在 x 和 y 均对称的视图体中，所指定的(eye$x$, eye$y$, eye$z$)点总是处在荧屏上图像的中心点处，这样就可以使用一系列的变换函数逐渐移动该点，从而实现对场景的全景扫描。

---

void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,
GLdouble centerx, GLdouble centery, GLdouble centerz,
GLdouble upx, GLdouble upy, GLdouble upz);

该函数定义一个视图矩阵并将其右乘到当前矩阵上，其结果返回给当前矩阵。期望的视点由参数 eyex, eyey, 和 eyez 指定，而参数 centerx, centery, 和 centerz 指定期望视线上的任意一点，通常取观测场景中心的点，这两点就确定了视线的方向。参数 upx, upy 和 upz 指定向上的方向（即视图体自下而上的方向）。

gluLookAt()是实用函数库的一部分，而且非常有用，但却不属于基本 OpenGL 库。这主要是因为它封装了几个基本的 OpenGL 函数——特别是 glTranslate*() 和 glRotate*()。为了说明这一点，我们假想一个置于任意视点并沿任意视线的照相机，其视点和视线均由 gluLookAt()来指定，同时原点处有一场景。要撤消 gluLookAt()所做的操作，需要做一系列的变换以使照相机位于原点并且指向 z 轴的负方向，即置于缺省的位置。具体的变换过程是这样的：首先将照相机向原点做简单的平移，然后在固定坐标系下做一系列绕三个坐标轴的旋转变换，最终使照相机朝向 z 轴的负方向。在 OpenGL 中允许沿任一坐标轴进行旋转，一个 glRotate*()函数就可以实现照相机任意的期望旋转。

如果要将一个任意的向量经过变换与另一向量重合（比如，z 轴的负方向），就需要做一些数学计算，其中旋转轴由两个归一化向量的叉积给出。将初始的两个向量归一化是为了得到旋转的角度，两个期望向量之间夹角的余弦值等于两个归一化向量的点积值。为了不发生混淆，由余弦确定的两个旋转角度值（x 度和 x+180 度）可以参考角度的正弦值来加以区分，这个正弦值等于两个归一化向量的叉积（参见附录 6 中对叉乘和点乘的定义）。

3. 创建用户的实用程序

对于某些特殊的应用，可能需要程序员自己定义一些特殊的变换子程序。由于这一技术很少用到并且难度也很大，这里只是留给读者练习。下面的练习给出两个常用的视图变换程序。

示例 1：假设程序员编制一个飞行仿真器的程序，目的是希望从飞行员的视点的角度能够显示整个场景。整个场景由原点位于跑道上的坐标系来描述，且飞机的坐标为（x,y,z）。这里我们进一步假设飞机具有一定的横倾角、纵倾角和航向角（均以飞机相对于重心的旋转角度给出）。

下面由多个视图变换组合的子程序可以实现上述目的。

```
void pilotView{GLdouble planex, GLdouble planey,
             GLdouble planez, GLdouble roll,
             GLdouble pitch, GLdouble heading)
{
     glRotated(roll, 0.0, 0.0, 1.0);
     glRotated(pitch, 0.0, 1.0, 0.0);
     glRotated(heading, 1.0, 0.0, 0.0);
     glTranslated(-planex, -planey, -planez);
}
```

示例 2：假设在某些应用程序中需要得到照相机绕位于原点的物体旋转的效果时，最好能使用极坐标来指定视图变换。在极坐标系下，用距离变量定义轨道的半径，或者说照相机离开原点的距离（照相机首先沿 z 轴的正方向移动）。用方位角描述照相机在 x-y 平面的旋转角度，以 y 轴的正方向为基线开始测量。类似的，仰角描述了照相机在 y-z 平面的旋转角度，以 z 轴的正方向为基线开始测量。最后，横摇角代表了视图体沿其视线的旋转轨迹。如下例程可实现上述的视图变换。

```
void polarView{GLdouble distance, GLdouble twist,
             GLdouble elevation, GLdouble azimuth)
{
     glTranslated(0.0, 0.0, -distance);
     glRotated(-twist, 0.0, 0.0, 1.0);
     glRotated(-elevation, 1.0, 0.0, 0.0);
```

```
    glRotated(azimuth, 0.0, 0.0, 1.0);
}
```

4.3 投影变换

上一节描述了如何构建模型视图矩阵以正确进行模型和视图变换操作。这一节将解释如何定义期望的投影矩阵，这一投影矩阵也用在场景中的顶点变换。在调用投影变换函数之前，需要先调用下面两个函数：

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

函数 glMatrixMode()的参数 GL_PROJECTION 指定投影矩阵为当前矩阵。由于每一个投影变换函数完整地描述了一个特定变换，所以不能把本节所讲述的各个投影变换函数混在一起使用。

投影变换的目的在于定义一个视图体，这个视图体有两个用途。它决定一个物体是如何投影到荧屏上的（透视投影法或正交投影法），而且确定了哪些物体或物体的哪些部分应该从最终的图像中裁剪掉。这里可以认为视点位于视图体的一端。

4.3.1 透视投影

关于透视投影的最显著的特征是透视的缩小：距离照相机越远的物体，在图像上就显得越小。这是因为在透视投影中，视图体是一个圆棱锥的平截台体，即是一个被平行于底面的平面截掉顶端的棱锥。透视就是把处在视图体内的物体向角锥的顶点（也就是照相机或视点所在处）投影。

同样一个物体，距离视点较近将比远离视点时在视图体中占据更大的比例，因此在平截台体的粗端将显得大一些。由于这种投影方法类似于人眼的视觉机制，所以通常用于动画、视景仿真等强调真实感的应用场合。

函数 glFrustum()定义一个平截台体，该函数实现透视投影矩阵的计算并将计算得到的矩阵左乘于当前投影矩阵（通常是单位矩阵）。我们已经知道，定义一个视图体是为了裁剪掉处在视图体之外的物体或物体的一部分，而平截台体的四个侧面加上顶面和底面，就相当于视图体的六个裁剪面，如图 4-11 所示。位于这些裁剪平面之外的物体或物体的部分将从图像中裁剪掉。

这里需要强调的是函数 glFrustum()并要求所定义的视图体是对称的。

---

void glFrustum(GLdouble left, GLdouble right, GLdouble bottom,
GLdouble top, GLdouble near, GLdouble far);

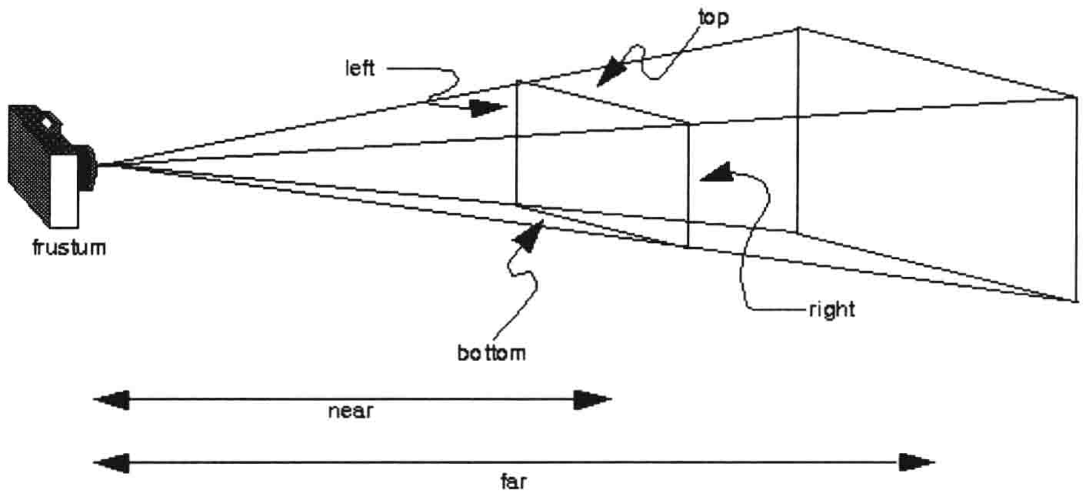


图 4-11　由 glFrustum()指定透视视图体

创建透视图的平截台体矩阵并左乘当前矩阵。平截台体的视图体由如下参数指定：参数(left, bottom, -near)和(right, top, -near)指定离视点近的裁剪平面（近裁剪平面）的左下角和右上角的坐标值(x, y, z)；参数 near 和 far 分别表示视点与近裁剪平面和远裁剪平面的距离。这些参数恒为正值。平截台体在三维空间有一个缺省的方向，可以通过对投影矩阵的旋转和平移来改变这一方向，但是这样做通常是不必要的，应该尽量避免。

在实际应用中，平截台体不必是对称的，而且平截台体的轴向也不必平行于 z 轴的方向。例如，可以使用函数 glFrustum()画一幅图像——这幅图就好像从一栋房子的矩形窗户看出去一样，此时窗户可以处在右上方。摄像师通常就是通过这种方式得到人工透视的效果。这种方法也可以用来得到分辨率更高的硬件计算图像，并可以打印输出。例如，如果希望得到两倍于屏幕分辨率的图像，可以将同一图像画四次，每次使用平截台体覆盖整个屏幕时，只用到四分之一的图像。在每一部分的图像经过渲染之后，就可以读取所有的像素点数据组成更高分辨率的图像（关于读取像素数据的内容请参见第 9 章）。

尽管函数 glFrustum()从概念上理解是很容易的，但本书并不推荐使用该函数来指定一个透视视图体。相反可以使用实用库的子程序函数 gluPerspective()，该函数可以创建一个与调用函数 glFrustum()所得到的同样形状的视图体，只不过指定的方式有点不同。在该函数中需要指定在 x-z 平面内视域的角度以及长宽比（x/y），而不是如函数 glFrustum()那样指定近裁剪面的各个角点坐标。这两个参数对于确定一个沿视线方向的棱锥体来说已经足够了，如图 4-12 所示，再加上分别指定视点到近裁剪平面和远裁剪平面的距离，这样就可以确定一个平截台体。虽然函数 gluPerspective()仅仅限于创建一个沿视线关于 x 和 y 轴均对称的平截台体，但这种平截台体是最常用的，所以，在很多实际应用中都采用函数 gluPerspective()。

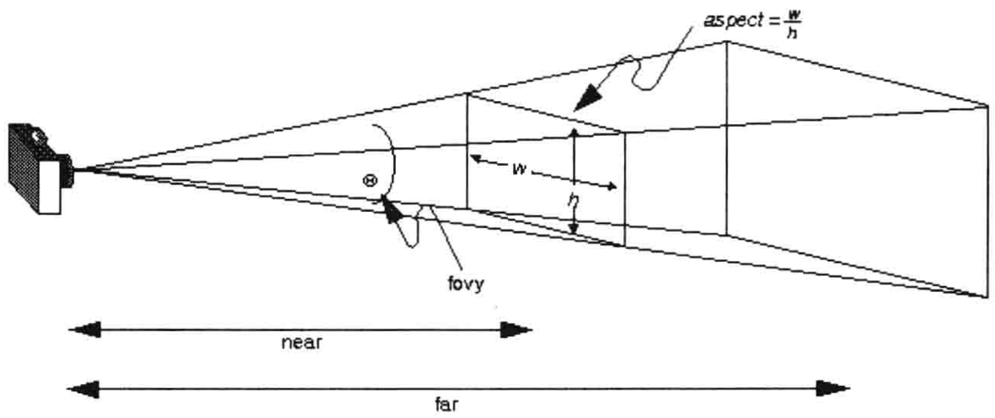


图 4-12　由 gluPerspective()指定的视图体

---

void gluPerspective(GLdouble fovy, GLdouble aspect,
　　GLdouble zNear, GLdouble zFar);

此函数创建一个对称透视图的平截台体矩阵，并将此矩阵左乘到当前矩阵上。参数 fovy 是在 x-z 平面内视区的角度，其值必须在区间[0.0，180.0]内。参数 aspect 为长宽比，是平截台体的宽度与高度之比。而参数 zNear 和 zFar 的值是视点（沿 z 轴负向）与两个裁剪平面的距离。所有这些参数恒为正值。

与函数 glFrustum()一样，可以使用旋转和平移来改变由函数 gluPerspective()创建的视图体的缺省方向。对于函数 gluPerspective()来说，其定义的视点为原点并且视线指向 z 轴的负方向。

使用函数 gluPerspective()时，需要选择合适的视野值，否则图像将发生变形。举个例子来说，假设场景将画在一个 11 英寸高的屏幕上，如果选择 90 度的视野，则眼睛必须离开屏幕 7.8 英寸时图像才不会扭曲变形（这是一个简单的几何问题）。如果眼睛离屏幕更远一些或更近一些，所得到的图像就会发生变形。如果图像的尺寸小于整个屏幕，则眼睛离屏幕就应该更近些。要得到合适的视野区域，需要计算出眼睛离屏幕的垂直距离以及窗口的大小，并且计算出该尺寸和距离对应的窗口扩展角度。这个值可能比想象中的要小一些。在变换的诊断这一章节中给出了得到合适视野值的计算方法。

前面提到的英寸和毫米等单位与 OpenGL 并没有任何关系。投影和变换实质上是没有单位的。至于说将近、远裁剪平面置于 1.0 和 20.0 米、英寸、公里还是里格，完全取决于个人习惯。惟一的规则是所有的测量单位应该一致，然后按照一定的比例画出图像。

4.3.2　正交投影

对于正交投影模式，视图体是一个成直角的平行六面体（即通常意义下的盒子，参见图 4-13）。

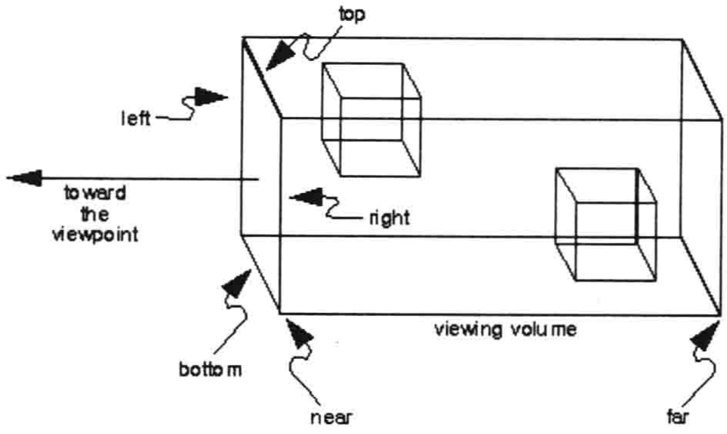


图 4-13　正交投影模式下的视图体

与透视投影不同，正交投影确定的视图体两端的尺寸大小相同，因此距照相机的远近并不影响物体显示的大小。这种投影模式通常应用在创建结构蓝图或计算机辅助设计当中，因为在这些应用中，保持物体的确切尺寸和它们之间的准确角度是至关重要的。

函数 glOrtho()用来创建正交的平行视图体。同使用函数 glFrustum()一样，程序员需要指定近裁剪平面的各个角点的坐标和两个裁剪平面的距离。

---

void glOrtho(GLdouble left, GLdouble right,
　　　　GLdouble bottom, GLdouble top,
　　　　GLdouble near, GLdouble far);

此函数创建一个正交平行视图体的矩阵，并左乘当前矩阵。近裁剪平面的矩形左下角的坐标为(left, bottom, -near)，而右上角的坐标为(right, top, -near)。而远裁剪平面的矩形左下角的坐标为(left, bottom, -far)，右上角的坐标为(right, top, -far)。

在没有其他变换的情况下，投影的方向平行于 z 轴的方向，而视点则朝向 z 轴的负方向。这也就意味着，如果裁剪平面在视点的前面，传递给参数 far 和 near 的值取 z 的负值；反之，取正值。

对于将一个二维的图像投影到二维屏幕上的特殊情形，要使用实用程序库的函数 gluOrtho2D()。除了场景中的 z 坐标值均被假定为在-1.0 到 1.0 之间以外，该函数与函数 glOrtho()的形式完全相同。如果使用二维顶点函数画二维的物体，所有的 z 坐标值均为 0，当然在 z 方向上也就没有裁剪这一说了。

---

void gluOrtho2D(GLdouble left, GLdouble right,
　　　　GLdouble bottom, GLdouble top);

创建一个矩阵，将二维坐标投影到屏幕上，并左乘当前矩阵。裁剪平面的左下角坐标为(left, bottom)，而右上角坐标为(right, top)。

4.3.3 视图体的裁剪

场景中所有物体的顶点经过模型、视图和投影变换之后，有的顶点位于视图体之内，而另一些顶点位于视图体之外，这些顶点将被裁剪掉。一般地，视图有六个裁剪面，分别是视图体的四个侧面和两个端面，当然，也可以在任何一个位置上设定额外的裁剪面，关于这一点请参见附加裁剪面。这里需要强调一点：对于被裁剪掉一边或多边的多边形，OpenGL 会重新自动地生成必要的边。

4.4 视区变换

我们首先回忆一下照相机类比法，我们知道视区变换是与选择照片尺寸这一任务相对应——最终想要得到的相片是钱包大小还是海报那么大呢？由于我们讨论的是计算机图形学，视区就是用来绘制图像的窗口中的一个矩形区域。图 4-14 显示了一个占据大部分屏幕的视区，这个视区是以窗口坐标来度量的，它反映了相对于窗口左下角的屏幕像素的位置。在该视区内的所有顶点的变换都是根据该点的模型视图矩阵和投影矩阵进行的，而视图体以外的顶点已被裁剪掉。

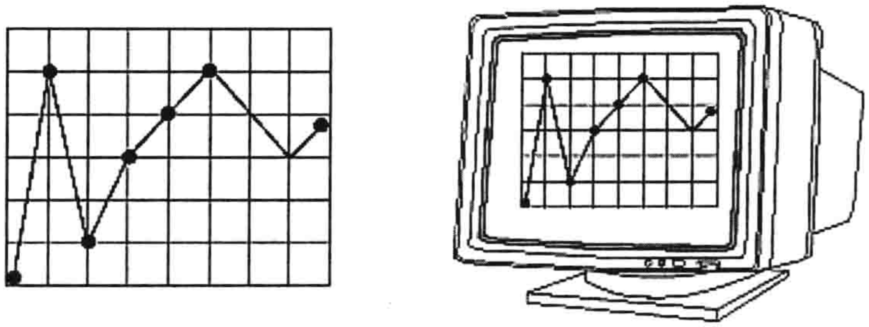


图 4-14 视区矩形

4.4.1 定义视区

在缺省情况下，视区设置成与所开窗口一样大小的矩形区域（以像素点为单位）。当然，可以用函数 glViewport()设置较小的绘画区域。例如，可以在同一个窗口里划分多个子窗口，为多个视图创造分屏效果。

---

void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);

此函数在窗口中定义一个矩形区域，并在该区域内进行图像映射。参数（x，y）指定为视区的左下角的坐标，而参数 width 和 height 为视区的尺寸。缺省情况下，视区的初始值为（0，0，winWidth，winHeight），其中，winWidth 和 winHeight 是当前窗口的尺寸。

通常，视区的长宽比应与视图体的长宽比相等。如果这两个比值不同，当投影的图像被映射到视区时将发生扭曲，如图 4-15 所示。在这之后改变窗口尺寸并不直接影响视区，因此，应用程序应该能够检测窗口尺寸改变这一事件，并且对视区进行适当的修改。

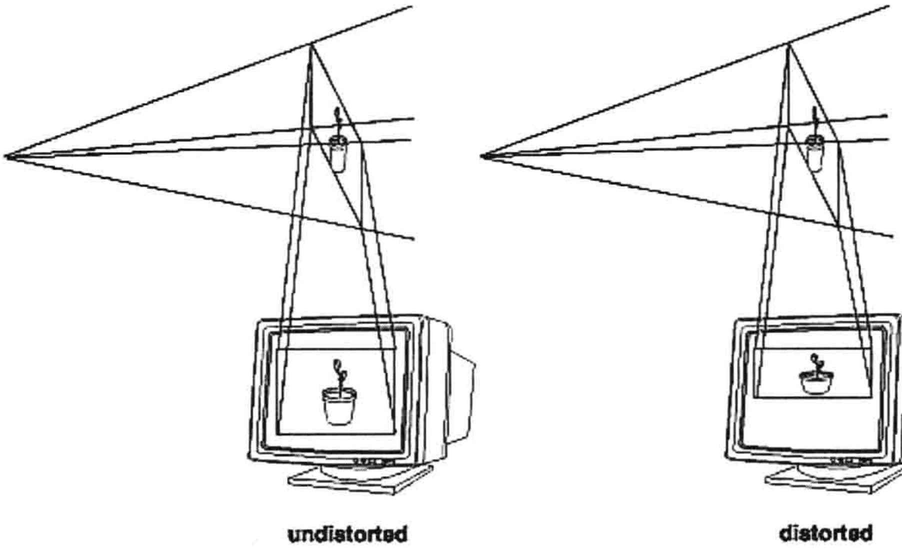


图 4-15　将视图体映射到视区

例如，下面的操作将一个正方形图像映射到正方形的视区中。

```
gluPerspective(myFovy, 1.0, myNear, myFar);
glViewport(0, 0, 400, 400);
```

然而，下面的操作将不等边矩形图像投影到正方形的视区中。因此，图像看起来好象是沿 x 轴做了压缩，如图 4-15 所示。

```
gluPerspective(myFovy, 2.0, myNear, myFar);
glViewport (0, 0, 400, 400);
```

为了避免扭曲，应该如下定义视区：

```
glViewport(0, 0, 400, 200);
```

试一试

修改现有的程序，将物体绘制两次，每一次在不同的视区中。可以在每个视区以不同的投影和/或视图变换绘制物体。如要创建两个并排的视区，可进行模型变换、视图变换和投影变换的同时，调用下列函数：

```
glViewport (0, 0, sizex/2, sizey);
........
glViewport (sizex/2, 0, sizex/2, sizey);
```

4.4.2 变换后的 Z 坐标

z 坐标，即深度坐标，该坐标在视区变换期间进行编码，然后存储。利用函数 glDepthRange() 可以将 z 坐标的值通过缩放操作变换到期望的范围之内。与窗口坐标 x、y 不同的是，OpenGL 在处理 z 坐标时，认为其变化范围总是在 0.0 到 1.0 内。

---

void glDepthRange(GLclampd near, GLclampd far);

在视区变换过程中，此函数为 z 坐标定义编码。参数 near 及 far 的值分别表示可以存储在深度缓存中的最小值和最大值。在缺省情况下（也是最常用的情况），参数值分别为 0.0 和 1.0；这些参数值截取在区间[0，1]之内。

4.5 变换的诊断

在实际生活中，让照相机的方向指向右侧是非常容易的，但在计算机图形学中，必须通过坐标和角度来指定位置和方向。我们可以证明，众所周知的黑屏效应其原理是非常简单的，但却实现不了。不过呢，有些事件发生错误后，常常是干脆在屏幕上什么也不画，从而导致这种黑屏效应。例如，相机未正确地对准场景，以及拍摄的模型在身后等等，都将导致这种结果。也就是说，在屏幕上打开的窗口中绝不会有任何东西画出。如果没有选择足够宽的视野来观察你的物体，或者视野太窄而使物体显得太大，都会产生类似的问题。

如果读者发现自己费了很大的力气来编程却仅仅创建出一个黑屏，可以试一试下面这些诊断步骤：

1）检查可能发生的一些明显的错误：确信系统是否通电；保证使用了不同于屏幕底色的颜色绘制物体；确信使用的状态（例如光照、纹理、alpha 混合、逻辑操作或锯齿现象的消除等）能够按期望的那样正确地开、关。

2）对于投影函数，参数 near 和 far 确定的坐标是从视点开始测量距离，并且缺省情况下是沿负 z 轴方向进行观察。因此，如果参数 near 的值为 1.0，参数 far 的值为 3.0，只有 z 坐标在-1.0 和-3.0 之间的物体才能看得见。为了确保仅裁剪掉场景外的部分而不是把所有的场景剪掉，可以先将参数 near 和 far 的值设置成无意义的值以得到所有处在场景中的物体是可视的，如 0.001 和 1000000.0。虽然这样做会对深度缓冲区和雾等操作产生负面影响，但却可以发现不小心裁剪掉的物体。

3）确定视点的位置，观察的方向，以及目标的位置。通过确定这些必要的参数值，可以帮助程序员创建真正的三维空间并计算出这些要素。

4）确定旋转的位置。对于全局坐标系而言，如果经过多次变换操作之后，物体不平移回原始位置，那么旋转的结果将难以预测。当然，如果不想绕原点旋转，绕任意点旋转也是

可以实现的。

5）明确目的。要么调用函数 gluLookAt()使物体位于视图体内，要么在原点附近绘制物体，并使用函数 glTranslate*()做视图变换，使照相机沿 z 方向移动到足够远的地方，从而使物体落在视图体内部。一旦这些物体已经可视，还应继续试着调整改变视图体，以求达到理想的结果。

即使照相机已经对准正确的方向，并且已经可以看到目标，还会出现目标太大或太小的情况。如果使用函数 gluPerspective()，则需要通过调整该函数中的第一个参数值来改变视野的角度。这里可以根据目标的尺寸及目标与视点的距离，通过三角函数计算视野角度值。通常的做法是期望角度一半的正切，就等于目标尺寸的一半与目标和视点的距离相除的结果（如图 4-16 所示）。因此，可以利用反正切函数来计算期望视角值的一半。例 4-3 采用了函数 atan2()给定三角形的邻边及对边，计算其反正切值。最后，需要将结果从弧度转化为度。

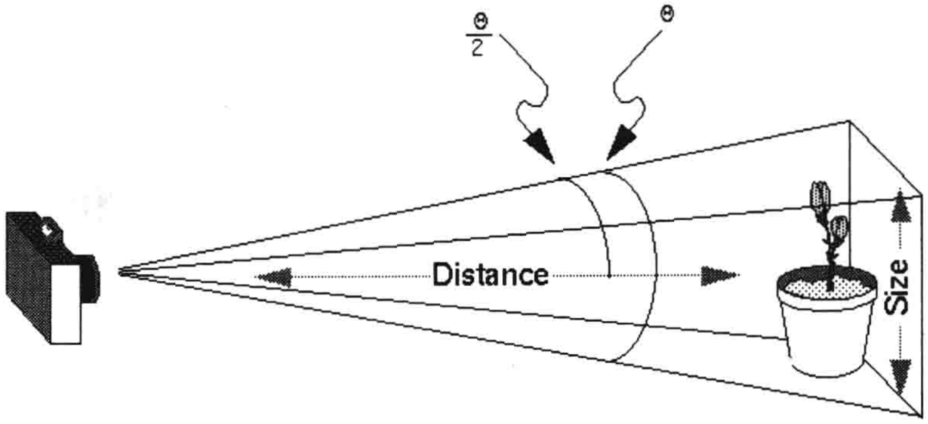


图 4-16　应用三角法计算视野

例 4-3　视野的计算

```
#define PI 3.1415926535
double calculateAngle(double size, double distance)
{
   double radtheta, degtheta;
   radtheta = 2.0 * atan2 (size/2.0, distance);
   degtheta = (180.0 * radtheta) / PI;
   return (degtheta);
}
```

当然，通常目标的精确尺寸是不知道的，并且只能确定视点与场景中的某一点之间的距离。为获得相当好的近似值，要通过确定场景中所有物体的 x, y, z 坐标的最大及最小值，来确定将场景围住的立方体盒（bounding box）的大小。然后计算此盒的内接球的半径，由球心确定距离，而由半径确定尺寸。

例如，假定物体所有坐标都满足方程-1≤x≤3，5≤y≤7 和–5≤z≤5，则盒的中心为（1，6，0），内接球半径是从盒心到其一角点距离，比如（3，7，5），其计算式子如下：

$$\sqrt{(3-1)^2+(7-6)^2+(5-0)^2}=\sqrt{30}=5.477$$

若视点是（8，9，10），它与该立体盒中心的距离是：

$$\sqrt{(8-1)^2+(9-6)^2+(10-0)^2}=\sqrt{158}=12.570$$

视野半角的正切值为 5.477/12.570（即 0.4357），因此，可得视野角约等于 47°。

视野角度的大小与视点位置之间的搭配关系将影响图像的真实感。大的视野角度如 179°，那么为了获得真实感，视点与图像之间的距离可能就只有零点几英寸大小。因此，如果计算出来的视野角度值太大，那么必须将视点远离目标或者让目标远离视点，从而减小视野角度的大小。

4.6 矩阵堆栈的操作

现在，经过程序员创建、加载并相乘后的模型视图矩阵和投影矩阵已被放置在它们各自堆栈的顶端：每一个矩阵都是矩阵堆栈中的最上面的成员（如图 4-17 所示）。

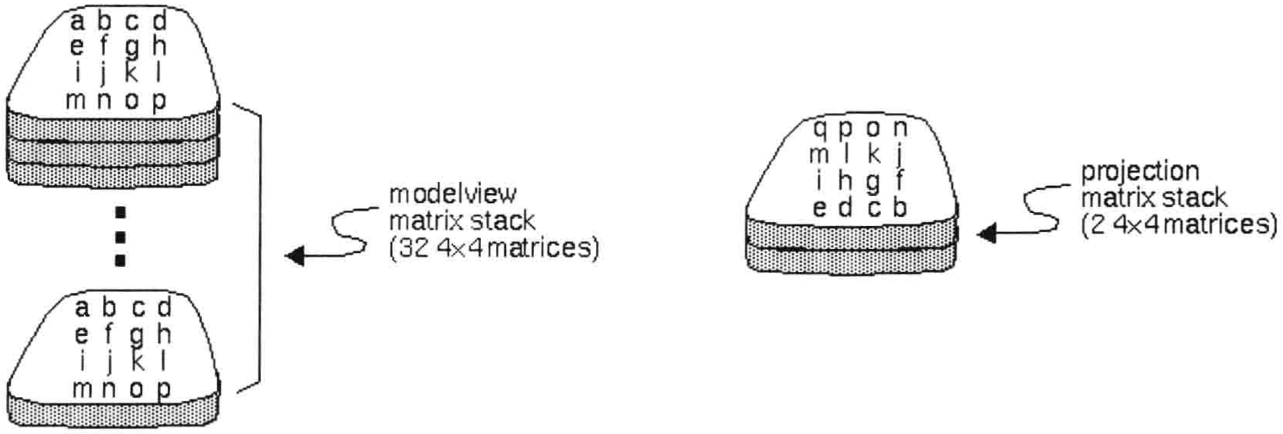


图 4-17 模型视图及投影矩阵堆栈

矩阵堆栈对构造分层模型是很有用处的，利用它可以把简单的物体构造成复杂的物体。比如：假设画一辆有四个车轮的汽车，而且每一车轮用五个螺钉与汽车固定。由于所有的车轮是相同的，所有的螺钉也都相同，因此可以用一个简单的程序画车轮，用另一个程序画螺钉。这些子程序以方便的位置和方位（即轴与 z 轴重合、以原点为中心）绘制车轮或螺钉。在绘制汽车，包括车轮及螺钉时，可以用不同的变换四次调用车轮绘制子函数，以便每次都可以将车轮定在准确的位置。在绘制车轮时，还要绘制 5 个螺钉，并将螺钉移动到车轮的适当位置。

绘制车身和车轮的大致步骤为：首先画车身，同时记住画完车身后的当前位置 A，然后平移到右前轮的位置上画出车轮；随后对上一个平移变换作反变换，即使得当前位置仍然为 A，记住该位置，然后将其平移到左前轮……

类似地，对每个要绘制的车轮，记住当前位置，然后将当前位置移到需要绘制的螺钉位置上，当然，在画下一个螺钉之前，必须对上一次绘制螺钉时所做的变换做一次反变换。

在上述过程中，之所以要做一次反变换，主要是为了利用某个基本物体的当前位置作为参考来绘制其他物体，这样可以保证概念更清晰，计算更容易。如在绘制车轮时，是以车身原点坐标作为参考坐标；在绘制螺钉时，车轮坐标则成为螺钉的参考坐标。

由于各种变换都是以矩阵方式存储的，对于连续的存储、变换、反变换的排序，矩阵堆栈提供一种理想的机制。所有描述过的矩阵操作，如函数 glLoadMatrix()，glMultMatrix()，glLoadIdentity()，以及创建特定的变换矩阵的函数等，负责处理当前矩阵，或栈顶矩阵。函数 glPushMatrix()和 glPopMatrix()执行堆栈操作，其中函数 glPushMatrix()确定位于栈顶的矩阵，即将当前矩阵压入栈顶，而函数 glPopMatrix()函数用来删除栈顶的矩阵。简单地说，函数 glPushMatrix()表示“记住你的当前位置”，而函数 glPopMatrix()表示“回到原处”。

---

void glPushMatrix(void);

此函数将所有矩阵压入由函数 glMatrixMode()确定的当前堆栈，而原来的栈顶矩阵复制到第二个矩阵中，第二个矩阵复制到第三个矩阵，依次类推。如果压入的矩阵太多，调用该函数将产生错误。

---

void glPopMatrix(void);

此函数从堆栈中弹出栈顶矩阵，原来第二层矩阵就成为栈顶矩阵。当然，当前堆栈由函数 glMatrixMode()确定。如果栈顶矩阵的内容被破坏，或者堆栈中只包含一个矩阵，调用函数 glPopMatrix()将产生错误。

假定已经有了绘制车身、车轮、以及螺钉的函数，在例 4-4 中将绘制一辆汽车。

例 4-4　矩阵的压入及弹出

```
draw_wheel_and_bolts()
{
  long i;
  draw_wheel();
  for(i=0;i<5;i++){
    glPushMatrix();
    glRotatef(72.0*i,0.0,0.0,1.0);
```

```
        glTranslatef(3.0,0.0,0.0);
        draw_bolt();
        glPopMatrix();
    }
}
draw_body_and_wheel_and_bolts()
{
  draw_car_body();
  glPushMatrix();
  lTranslatef(40,0,30);               /* 移到第一轮的位置 */
  draw_wheel_and_bolts();
  glPopMatrix();
  glPushMatrix();
  glTranslatef(40,0,-30);                  /* 移到第二轮的位置 */
  draw_wheel_and_bolts();
  glPopMatrix();
  ...                 /* 以类似方法绘制后面的两个轮子 */
}
```

这段代码假定车轮和螺钉的轴线与 z 轴一致，螺钉与车轮中心距离为 3 个单位（可以为英寸），螺钉之间的间隔为 72 度，前轮布置在车身原点前面 40 个单位，左右车轮距原点 30 个单位。

与单个矩阵相比，应用堆栈将更加有效，尤其是堆栈以硬件方式实现的情况下。当压入一个矩阵时，不必立即将当前数据复制回主进程，硬件将一次复制许多矩阵元素。有时，可能更希望将单位阵保存在栈底 ，这样就不必反复地调用 glLoadIdentity()函数。

4.6.1 模型视图矩阵堆栈

如前所述，模型视图矩阵是视图变换矩阵和模型变换矩阵乘积的累计结果。每个视图或模型变换都要产生一个新的矩阵，它与当前的模型视图矩阵相乘，相乘的结果成为新的当前矩阵，也就是说，模型视图矩阵是多次变换的合成。模型视图矩阵堆栈至少包含 32 个 4 阶方阵，其中栈顶矩阵的初始状态是单位阵。当然，OpenGL 支持多于 32 个矩阵的堆栈，程序员可以用 GL_MAX_MODELVIEW_STACK_DEPTH 作为参数，通过查询函数 glGetIntegerv()找出堆栈中最大允许的矩阵数。

4.6.2 投影矩阵堆栈

投影矩阵包含一个用于投影变换的矩阵，该矩阵描述了视图体。通常，并不希望对投影

矩阵进行合成，因此在进行投影变换之前应该调用函数 glLoadIdentity()。同样的原因，投影矩阵堆栈一般仅需要两级深度，但在特殊情况下也允许多于两个的 4 阶方阵。通过调用带参数 GL_MAX_PROJECTION_STACK_DEPTH 的函数 glGetIntegerv()可以得到堆栈的深度。

在某些应用程序中，既要在窗口中显示三维场景，同时又要显示带有文本的帮助窗口，这时一般要用到堆栈中的第二个矩阵。因为文本最容易用正交投影绘制，因此可以先存储当前的透视投影矩阵，然后采用正交投影方式显示帮助信息，最后返回到以前的投影方式。

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();                        /* 存储当前投影矩阵 */
glLoadIdentity();
glOrtho(...);                          /* 设置显示帮助 */
display_the_help();
glPopMatrix();
```

当然，程序员可以创建自定义的投影矩阵，用来进行任意的投影变换。例如，对于两点透视法，OpenGL 及其实用库并没有嵌入相应的实现机制。然而，如果程序员想得到类似于草稿方式绘图的效果，就需要构造对应于两点透视方法的投影矩阵，这时候，就需要用户自己定义该投影矩阵。

4.7　附加裁剪平面

除了视图体的 6 个裁剪平面（左，右，下，上，近，远）之外，程序员可以定义 6 个附加的裁剪平面，以便更严格地限制视图体，如图 4-18 所示。这种方法对于删除场景中无关的物体非常有用，比如说要显示某个物体部分被切除后的视图。

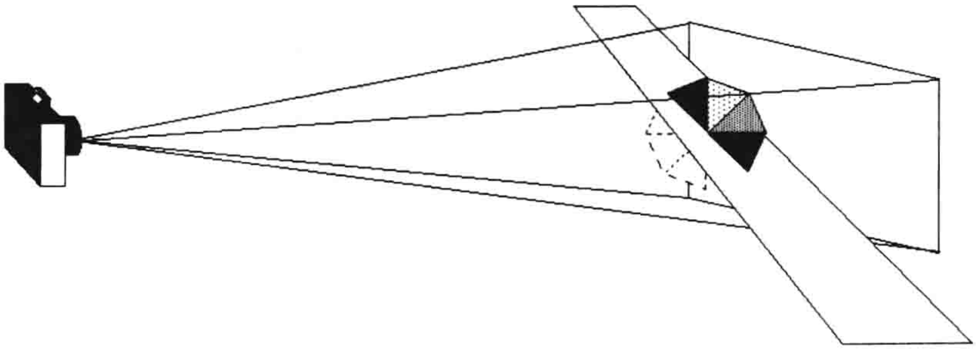


图 4-18　附加裁剪平面和视图体

每个平面由方程 $Ax+By+Cz+D = 0$ 的系数确定。裁剪平面根据模型变换和视图变换自动地进行适当的变换。裁剪结果为视图体和所有附加裁剪平面定义的半空间的交集，而对于被裁剪掉一些边的多边形，OpenGL 将为其生成必要的附加边。

void glClipPlane(GLenum plane, const GLdouble *equation);

此函数定义一个裁剪平面。参数 equation 为指向平面方程 Ax+By+Cz+D = 0 的 4 个系数的指针。所有满足

$(A\ B\ C\ D)M^{-1}(x_e\ y_e\ z_e\ w_e)^T \geqslant 0$

的人眼坐标 (x_e, y_e, z_e, w_e)的点都位于该平面定义的半空间中，而该半空间以外的所有点都被裁剪掉。其中，M 是调用函数 glClipPlane()时当前的模型视图矩阵，参数 plane 为 GL_CLIP_PLANEi，i 是 0 到 5 之间的整数，指定在 6 个附加裁剪平面中被定义了哪几个平面。

作为函数 glClipPlane()的执行结果所做的裁剪是在人眼坐标中，而不是在裁剪坐标中完成的。当投影矩阵为奇异阵时尤其应注意到这种差别，因为即使投影矩阵是奇异阵的时候，三维情况下的裁剪也是在人眼坐标中进行的。

函数 glEnable(GL_CLIP_PLANEi)激活所定义的附加裁剪平面；函数 glDisable(GL_CLIP_PLANEi)取消指定的附加裁剪平面。

OpenGL 支持附加裁剪平面多于 6 个的操作，调用函数 glGetIntegerv()，使用参数 GL_MAX_CLIP_PLANES 可以得到 OpenGL 所能支持的附加裁剪平面数。

一个附加裁剪平面的例程

例 4-5 给出了有两个附加裁剪平面的球体线框图的绘制方法，其结果是该球体被切除了 3/4，如图 4-19 所示。

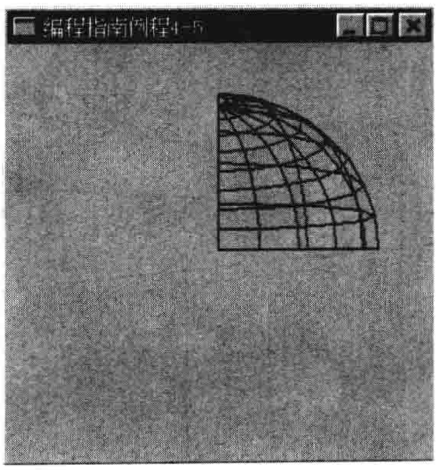


图 4-19 裁剪后的球体线框图

例 4-5 具有两个附加裁剪平面的球体的线框图：clip.c

```
#include <windows.h>
```

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void CALLBACK display(void)
{
   GLdouble eqn[4] = {0.0, 1.0, 0.0, 0.0};    /* y <0  */
   GLdouble eqn2[4] = {1.0, 0.0, 0.0, 0.0};   /* x <0  */
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f (0.0, 0.0, 0.0);
   glPushMatrix();
   glTranslatef (0.0, 0.0, -5.0);
    glClipPlane (GL_CLIP_PLANE0, eqn);
   glEnable (GL_CLIP_PLANE0);
   glClipPlane (GL_CLIP_PLANE1, eqn2);
   glEnable (GL_CLIP_PLANE1);
   glRotatef (90.0, 1.0, 0.0, 0.0);
   auxWireSphere(2.0);
   glPopMatrix();
   glFlush();
}

void myinit (void)
{
    glClearColor(192.0/255.0, 192.0/255.0, 192.0/255.0, 1.0);
   glShadeModel (GL_FLAT);
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
   glMatrixMode(GL_MODELVIEW);
```

```
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例程 4-5");
    myinit ();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
     return 0;
}
```

试一试

- 试着改变例 4-5 中描述裁剪平面的系数。
- 试着调用一个模型变换函数，如 glRotate*()，看其如何影响函数 glClipPlane()的执行结果。试着独立于物体之外移动场景中的裁剪平面。

4.8 组合变换举例

本节将具体说明结合几种变换以实现某种特殊效果的方法。这里通过讨论两个例子来说明问题：一个是建立太阳系模型，其中的物体除了自转之外同时还要公转；另一个是多关节的机器人手臂模型，通过有效地坐标变换来描述各个关节的相对运动。

4.8.1 建立一个太阳系模型

本节的程序绘制一个简单的太阳系，其中有一个太阳，一个行星，均用相同的球体函数绘制。要编写这个程序，调用函数 auxSphere()绘制球体，这里通过指定合适的参数来确定两个球体的各自大小，同时需要使用函数 glRotate*()来实现行星绕太阳的公转和绕自身轴的自转，还需要使用函数 glTranslate*()实现行星沿其轨迹的移动。

要绘制太阳系，首先需要设置一个投影变换和一个视图变换。在本例中，使用了函数 gluPerspective()和 gluLookAt()来指定这两个变换。绘制太阳非常简单，只要将其放置在固定坐标系的原点即可，而这恰好是绘图函数缺省的位置。因此，绘制太阳并不需要平移，同时也可以使用函数 glRotate*()让太阳绕任意轴转动。如图 4-20 所示，为了绘制绕太阳转动的行星，需要进行几次模型变换。行星每天要完成一次绕自身轴的自转，每年要完成一次绕太阳的公转。

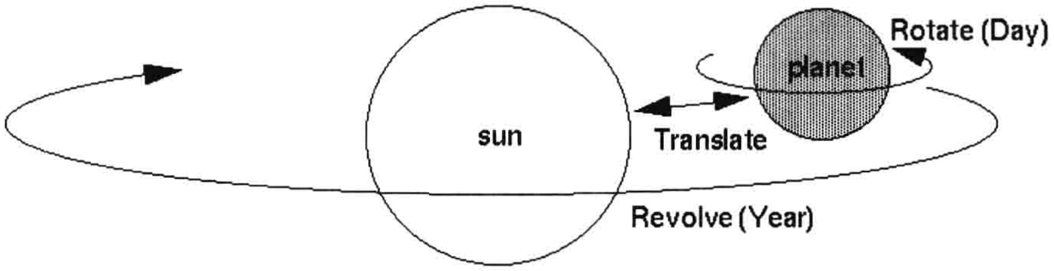


图 4-20　太阳系

为了确定模型变换的顺序，可以设想一下在局部坐标系中进行的操作。第一个函数 glRotate*()将使初始与固定坐标重合的局部坐标发生转动。接下来，函数 glTranslate*()将局部坐标移动到行星轨道上的某个位置，所移动的距离应与轨道的半径相等。因此，第一个函数 glRotate*()实际上确定了行星位于轨道的什么位置（或一年中的什么时间）。

第二个函数 glRotate*()使局部坐标系绕局部轴转动，由此确定了行星所处的一天中的时间。一旦调用了所有这些变换函数，行星便可以绘制出来。

下面所列的是绘制太阳和行星的 OpenGL 函数的总结。程序的全部清单如例 4-6 所示。

```
glPushMatrix();
auxWireSphere(1.0);                /* 绘制太阳 */
glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
glTranslatef (2.0, 0.0, 0.0);
glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
auxWireSphere(0.2);               /* 绘制小行星 */
glPopMatrix();
```

例 4-6　一个行星系：planet.c

```
#include <windows.h>

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

static int year = 0, day = 0;

static void CALLBACK dayAdd(void)
{
   day = (day + 10) % 360;
}
```

```
static void CALLBACK daySubtract(void)
{
   day = (day - 10) % 360;
}

static void CALLBACK yearAdd(void)
{
   year = (year + 5) % 360;
}

static void CALLBACK yearSubtract(void)
{
   year = (year - 5) % 360;
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 0.0, 0.0);
    glPushMatrix();
    auxSolidSphere(1.0);        /* 绘制太阳 */
    glRotatef((GLfloat) year, 0.0, 1.0, 0.0);
    glTranslatef (2.0, 0.0, 0.0);
    glRotatef((GLfloat) day, 0.0, 1.0, 0.0);
    glColor3f (0.0, 0.0, 1.0);
    auxSolidSphere(0.2);        /* 绘制小行星 */
    glPopMatrix();
    glFlush();
}

void myinit(void)
{
    glShadeModel(GL_FLAT);
}
```

```
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0,(GLfloat)w/(GLfloat)h,1.0,20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (0.0, 0.0, -5.0);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
    auxInitPosition(0, 0, 500, 500);
    auxInitWindow("编程指南例程 4-6");
    myinit();
    auxKeyFunc(AUX_LEFT, yearSubtract);
    auxKeyFunc(AUX_RIGHT, yearAdd);
    auxKeyFunc(AUX_UP, dayAdd);
    auxKeyFunc(AUX_DOWN, daySubtract);
    auxReshapeFunc(myReshape);
    auxMainLoop(display);

    return 0;
}
```

试一试

- 试着为行星加上月亮。或者加上多个行星，每个行星多个月亮。提示：利用函数 **glPushMatrix()**和 **glPopMatrix()**在适当的时刻保存和恢复坐标系的位置和方位。如果要绘制围绕一个行星的多个月亮，需要在每个月亮的位置确定之前保存坐标系，在每个月亮被画出后恢复坐标系。
- 试着将行星的轴倾斜。

4.8.2 建立一个有关节的机器人手臂模型

本节讨论一个用于创建有两个或多个关节的机器人手臂的程序。机器人手臂由肩、肘或

其他关节处的节点连接而成。图 4-21 显示了机器人手臂的一个关节。

可以将一个经过缩放之后的长方体作为机器人手臂的一段，但首先必须对每一段作恰当的模型变换。因为局部坐标的原点初始位置在这段长方体的中心，因此需要将局部坐标移到其一端。否则，各个分段将绕中心而不是绕节点转动。

在调用函数 glTranslate*()创建节点并调用函数 glRotate*()使长方体绕支点转动之后，应该将局部坐标移回长方体中心，然后再绘制这个长方体。下面是用于绘制机器人手臂第一段的代码（程序的全部清单见例 4-7）。

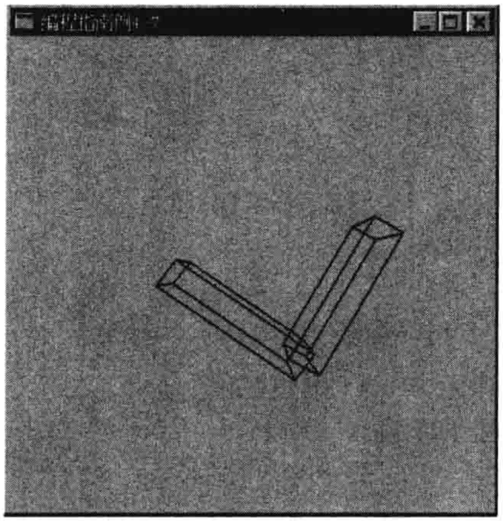


图 4-21　机器人手臂的一段

```
glTranslatef (-1.0, 0.0, 0.0);
glRotatef ((GLfloat) shoulder_angle, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
auxWireBox(2.0, 0.4, 1.0);
```

为了创建第二段机器人手臂，需要将局部坐标系移动到下一个节点。因为坐标系已被旋转过，此时 x 轴与旋转过的机器人手臂的长度方向一致。因此，沿 x 轴的平移就将局部坐标系移动到下一个节点。一旦移动到了该点，就可以利用类似的方法绘制第二段机器人手臂。其他各段机器人手臂也都可以利用类似的方法绘制。图 4-22 显示一有手指的机器人手臂。

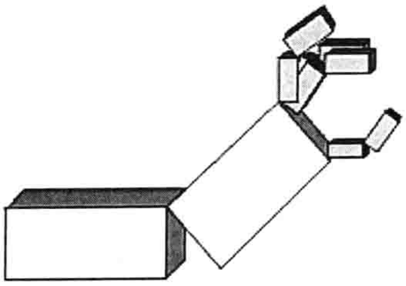


图 4-22　有手指的机器人手臂

```
glTranslatef (1.0, 0.0, 0.0);
glRotatef ((GLfloat) elbow_angle, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
auxWireBox(2.0, 0.4, 1.0);
```

例 4-7　一个机器人手臂：robot.c

```
#include <windows.h>

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

static int shoulder = 0, elbow = 0;

static void CALLBACK elbowAdd (void)
{
    elbow = (elbow + 5) % 360;
}

static void CALLBACK  elbowSubtract (void)
{
    elbow = (elbow - 5) % 360;
}

static void CALLBACK  shoulderAdd (void)
{
    shoulder = (shoulder + 5) % 360;
}

static void CALLBACK  shoulderSubtract (void)
{
    shoulder = (shoulder - 5) % 360;
}

static void CALLBACK  display(void)
{
```

```
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (0.0, 0.0, 0.0);
    glPushMatrix();
        glTranslatef (-1.0, 0.0, 0.0);
        glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
        glTranslatef (1.0, 0.0, 0.0);
        auxWireBox(2.0, 0.4, 1.0);      /* 肩 */
        glTranslatef (1.0, 0.0, 0.0);
        glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
        glTranslatef (1.0, 0.0, 0.0);
        auxWireBox(2.0, 0.4, 1.0);      /* 肘 */
    glPopMatrix();
    glFlush();
}

void myinit (void)
{
    glClearColor(192.0/255.0, 192.0/255.0, 192.0/255.0, 1.0);
    glShadeModel(GL_FLAT);
}

static void CALLBACK  myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(65.0,(GLfloat)w/(GLfloat)h,1.0,20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -5.0);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
```

```
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow ("编程指南例 4-7");
    myinit ();
    auxKeyFunc (AUX_LEFT, shoulderSubtract);
    auxKeyFunc (AUX_RIGHT, shoulderAdd);
    auxKeyFunc (AUX_UP, elbowAdd);
    auxKeyFunc (AUX_DOWN, elbowSubtract);
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
     return 0;
}
```

试一试

- 修改例 4-7，为机器人手臂添加其他的关节。
- 修改例 4-7，在同一位置添加多个机器人关节。例如，在腕部为机器人手臂加上“手指”，如图 4-22 所示。提示：利用函数 glPushMatrix() 和 glPopMatrix()保存和恢复腕部的坐标系的位置和方位。如果要在腕部绘制“手指”，必须在每个手指的定位之前保存当前矩阵，在每个手指画出后，恢复当前矩阵。

第 5 章　显示列表

本章主题

显示列表是存储起来用于稍后执行的一组 OpenGL 命令。激活一个显示列表后，就按照显示列表中预先排好的次序执行其存储的命令。大部分的 OpenGL 命令既可存储在显示列表中也可在直接模式中直接发送，直接发送的命令将被立即执行。在程序中可以自由地混合使用直接模式编程和显示列表。本书前面给出的程序示例均采取直接模式。本章将讨论什么是显示列表，以及如何更好地使用它们。本章包括如下几个主要部分：

- 一个使用显示列表的范例，给出了一个简短的例子，例举了用于显示列表的几个基本命令。
- 显示列表设计准则，解释了在何种场合使用显示列表。
- 创建和执行显示列表，详细讨论了用于创建和执行显示列表的 OpenGL 命令。
- 管理显示列表及其索引，详细讨论了如何让 OpenGL 为程序员自动生成显示列表索引。
- 执行多个显示列表，论述了如何连续执行多个显示列表，并用一个小的字集合示例加以说明。
- 封装模式的转变，讨论了如何利用显示列表在不同的模式之间进行高效切换。

5.1　一个使用显示列表的范例

显示列表是一种对 OpenGL 命令集进行命名和组织的一种方便高效的方式。例如，假如要用 100 条线段来画一个圆。如果不用显示列表的方式，在直接模式当中需要编写如下程序代码：

```
drawCircle()
{
  GLint i;
  GLfloat cosine, sine;
  glBegin(GL_POLYGON);
    for(i=0;i<100;i++){
      cosine=cos(i*2*PI/100.0);
      sine=sin(i*2*PI/100.0);
```

```
        glVertex2f(cosine,sine);
      }
    glEnd();
}
```

由于在每次渲染圆的过程当中都必须执行三角函数计算，可以想象这种方法效率将是非常低的。通过将所有的坐标存储在一个表中，然后在每次需要时从表中调用，便可以使上述低效的情况大大改观。

```
drawCircle()
{
  GLint i;
  GLfloat cosine, sine;
  static GLfloat circoords[100][2];
  static GLint inited=0;
  if(inited==0){
    inited=1;
    for(i=0;i<100;i++){
      circcoords[i][0]=cos(i*2*PI/100.0);
      circcoords[i][1]=sin(i*2*PI/100.0);
    }
  }
  glBegin(GL_POLYGON);
    for(i=0;i<100;i++)
      glVertex2fv(&circcoords[i][0]);
  glEnd();
}
```

使用改进后的方法，仍然存在变量 i 的递增和判断的问题。我们所期望的是一次性绘制圆，并设法使 OpenGL 记住其画法以备后用。这便是设计显示列表的目的，如例 5-1 所示。

例 5-1　创建显示列表

```
#define MY_CIRCLE_LIST 1
buildCircle()
{
  GLint i;
  GLfloat cosine, sine;
  glNewList(MY_CIRCLE_LIST, GL_COMPILE);
    glBegin(GL_POLYGON);
```

```
        for(i=0;i<100;i++){
            cosine=cos(i*2*PI/100.0);
            sine=sin(i*2*PI/100.0);
            glVertex2f(cosine,sine);
        }
      glEnd();
    glEndList();
}
```

注意：用于画圆的代码都放在 glNewList()命令和 glEndList()命令之间。正是由这两个命令定义了显示列表。函数 glNewList()的变量 MY_CIRCLE_LIST 是一个整形索引量，用来惟一确定显示列表。然后利用命令 glCallList()便可以执行显示列表。

在显示列表中只能包括 OpenGL 调用。在例 5-1 中的其他调用，比如 C 语言函数 cos() 和 sin()并不存储在显示列表中。而坐标和其他变量（比如数组的内容）将在列表编译过程中进行求值并复制到显示列表中。显示列表在编译完成之后，其值就不能再改变。显示列表可以被删除，也可以重新创建，但不能对现存的显示列表进行编辑。

5.2 显示列表设计准则

OpenGL 的显示列表是为优化性能而设计的，尤其是优化网络上的应用程序，但它并不会降低本地机的性能。显示列表驻留在 OpenGL 服务器中，这在网络环境中与主机（或客户机）是不同的。在 OpenGL 的定义中详细讨论了 OpenGL 的客户/服务器模式。

为了优化性能，OpenGL 的显示列表是命令的缓存而不是动态数据库。换句话说，一旦一个显示列表创建之后，就是不可改变的。如果显示列表是可修改的，那么其性能将因为额外的显示列表搜索和执行内存管理而降低。当可修改的显示列表部分被改变时，内存的分配和释放将可能造成内存碎片。通常的情况，使用显示列表至少应该与使用直接模式一样快。因为显示列表驻留在服务器中，将使网络数据交换量降到最低程度，所以通过网络来发送 OpenGL 命令时，显示列表可以充分提高性能。

即使对于本地机，显示列表也有很高的效率，因为创建时，可以充分考虑显示列表与图形硬件之间更好的兼容。实现这种优化的特殊命令通常因 OpenGL 具体实现而异。例如，像 glRotate*()这样一个简单的命令如果放到显示列表中，其性能将显著提高，这是由于生成旋转矩阵的计算量很大（包括平方根运算和三角函数运算）。但是，在显示列表中只存储最后的旋转矩阵，所以显示列表执行的旋转命令可以达到硬件执行 glMultMatrix()同样的速度。一个复杂的 OpenGL 实现甚至可以将相邻的变换命令连接成一个矩阵乘法。

尽管不能保证对任何特殊应用显示列表都能在 OpenGL 具体实现中起到优化的作用，但可以保证显示列表的执行不会比直接执行其中的命令更慢。当然，显示列表会引进一些额外

的开销，如果一个列表不大，其开销可能超过其带来的优越性。下面列出了用于优化的可能的操作，并同时给出了讨论相应主题的章节。

- 矩阵操作（第 4 章）。大部分的矩阵操作要求 OpenGL 进行求逆计算。计算矩阵及其逆阵均由一个特殊的 OpenGL 命令将其存储在显示列表中。
- 控制位图和图像（第 5 章）。你所指定的光栅数据格式不可能象硬件那样理想。当显示列表进行编译时，OpenGL 可以将数据转变成硬件期望的表示法。由于字符串通常由一系列的小位图构成，这对提高光栅字符的绘制速度有显著的影响。
- 光照、材质及光照模型（第 7 章）。当在复杂的光照条件下绘制场景时，可能需要改变场景中的物体的材质。材质的设置涉及大量的计算，所以速度很慢。如果将材质定义加到显示列表中去，那么就不必在选择材质时反复进行计算，而只需存储计算结果；这样，渲染光照场景就快多了。关于如何利用显示列表来更改光照条件等值的详细信息，请参见“封装模式的变换”一节。
- 纹理（第 10 章）。因为硬件的纹理格式可能会与 OpenGL 的格式有所不同，如果将定义的纹理编译到显示列表中，那么这种转换就可在显示列表的编译过程中进行而不必在显示时进行，从而获得较高的效率。
- 多边形点划模式（第 3 章）。

这里所列出的指定属性的命令是与内容表密切相关的，为确保得到最优的性能，必须考虑到这一点。类似的情况还包括像素转换函数、光照模型以及纹理。由于这些主题都将在以后的章节中详细论述，因此这里不作详细讨论。下面的例子中人为地造成了通常不太可能出现的情况，来说明后续章节中所讨论问题的一个重要原则。

我们设想 OpenGL 实现中，在将矩阵存储到显示列表之前，执行了顶点的矩阵变换操作。这样，转换就是在显示之前，而不是在显示中进行。现在假设源代码如下：

```
glLoadMatrix(M);
glNewList(1, GL_COMPILE);
draw_some_geometric_objects();
glEndList();
```

在利用矩阵 M 进行转换之后，物体的顶点将被编译到显示列表中。假设显示列表已经按如下步骤激活：

```
glLoadMatrix(N);
glCallList(1);
```

在这种情况下，几何物体应该由矩阵 N 绘出，而显示列表中的数据已经由矩阵 M 在存储之前变换了。因此，显示列表必须存储原始数据的双份拷贝（包括未变换的和已变换的顶点），这是对内存的浪费。而且，有些顶点只需变换一次，却变换了两次。相反，如果如下定义显示列表，那样额外的数据就不必存储，也可做到完全优化。当然，在第二种情况，需要确保矩阵 M 就是所期望的变换矩阵。

```
glNewList(1, GL_COMPILE);
glLoadMatrix(M);
draw_some_geometry();
glEndList();
```

显示列表也有一些不足之处。比如，在例 5-1 中的 buildCircle()需要存储至少 200 个浮点数，而在直接模式中原始 drawCircle()的对象代码就可能小得多。另一个缺点是显示列表内容的不变性。为了优化性能，OpenGL 的显示列表内容是不可修改的，而且其内容是不可读的。

5.3 显示列表的创建和执行

如上所述，glNewList() 和 glEndList()可以用来定义显示列表的开始和结束，然后通过识别该列表的索引以 glCallList()进行激活。在例 5-2 中，显示列表是在 makeList()函数中创建的。显示列表中包括绘制一红色矩形的 OpenGL 命令。然后，在 display()例程中，显示列表被执行了 10 次，最后再以直接模式绘制一条直线。注意，显示列表要为命令以及必要的变量值分配内存。

例 5-2 使用显示列表：list.c

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
GLuint listName = 1;
void myinit (void)
{
    glNewList (listName, GL_COMPILE);
    glColor3f(1.0, 0.0, 0.0);
    glBegin (GL_TRIANGLES);
        glVertex2f (0.0, 0.0);
        glVertex2f (1.0, 0.0);
        glVertex2f (0.0, 1.0);
    glEnd ();
    glTranslatef (1.5, 0.0, 0.0);
    glEndList ();
    glShadeModel (GL_FLAT);
}
```

```
static void CALLBACK drawLine (void)
{
    glBegin (GL_LINES);
        glVertex2f (0.0, 0.5);
        glVertex2f (15.0, 0.5);
    glEnd ();
}

static void CALLBACK display(void)
{
    GLuint i;
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 1.0, 0.0);
    for (i = 0; i < 10; i++)
        glCallList (listName);
    drawLine ();
    glFlush ();
}

static void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
      gluOrtho2D (0.0, 2.0, -0.5 * (GLfloat) h/(GLfloat) w,1.5 * (GLfloat)
h/(GLfloat)w);
    else
      gluOrtho2D (0.0, 2.0 * (GLfloat) w/(GLfloat) h, -0.5, 1.5);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
```

```
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 400, 50);
    auxInitWindow ("编程指南例 5-2");
    myinit ();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
     return 0;
}
```

在显示列表中，函数 glTranslatef()用来改变要绘制的下一个物体的位置。如果不用该函数，每次调用显示列表将仅把三角形重画一次。在直接模式中调用的 drawLine()函数也将受到在它之前调用了 10 次的 glTranslatef()函数的影响。

注意：在显示列表中调用变换命令时，一定不要忘记考虑该命令对程序的后续部分的影响。

显示列表一次只能创建一个。换句话说，在创建另一个显示列表之前，程序员必须用 glEndList()来结束由 glNewList()创建的显示列表。如果并未创建显示列表，那么 glEndList()调用将会产生错误：GL_INVALID_OPERATION。

---

void glNewList (GLuint list, GLenum mode);

指明一个显示列表的开始。除了那些受限制的不能存储的函数，依次调用的 OpenGL 函数（直到调用 glEndList()来结束显示列表之前）都将被存储在显示列表中。参数 list 是一个标识该列表的惟一确定的正整数。参数 mode 的可能值为 GL_COMPILE 和 GL_COMPILE_AND_EXECUTE。如果希望接下来的 OpenGL 命令仅放在显示列表中而不执行，可以使用 GL_COMPILE；如果希望这些命令立即执行然后存储在列表中以备后用，则使用 GL_COMPILE_AND_EXECUTE。

---

void glEndList (void);

标志显示列表的结束。

5.3.1 显示列表的存储内容

在创建一个显示列表时，在显示列表中存储的只是表达式的值。因此，如果在后面的执行中数组的值发生了改变，显示列表的值并不会改变。在下面的这段代码中，显示列表包含了一条将当前颜色设置为黑色（0.0，0.0，0.0）的命令。后来虽然向量 color_vector 的值变为红（1.0，0.0，0.0），但对显示列表并没有产生影响，因为显示列表中包含的值在其创建

时就一直有效。

```
GLfloat color_vector[3]={0.0,0.0,0.0};
glNewList(1,GL_COMPILE);
glColor3fv(color_vector);
glEndList();
color_vector[0]=1.0;
```

OpenGL 中有一些命令不能存储在显示列表中，而是在显示列表内部执行。一般情况下，通过参量传递参数或有返回值的命令都不能在显示列表中存储，因为列表可能在参数的原始定义范围之外调用。如果在创建显示列表时调用这些命令，它们将马上执行而并不会存储在显示列表中。下面是不能存储在显示列表中的 OpenGL 命令（特别需要注意的是，如果在创建一个显示列表时再次调用 glNewList()，将产生错误）。

```
glDeleteLists()
glIsEnabled()
glFeedbackBuffer()
glIsList()
glFinish()
glPixelStore()
glFlush()
glReadPixels()
glGenLists()
glRenderMode()
glGet*()
glSelectBuffer()
```

为了进一步说明为什么这些命令不能存储在显示列表中，我们考虑通过网络使用 OpenGL，客户和服务器分置的情况。在创建一个显示列表后，由于它驻留在服务器中，服务器便不能依赖客户机中任何关于显示列表的信息。如果在显示列表中允许有 glGet*()和 glIs*()等查询命令，那么调用程序就不知道该如何处理从网络返回的数据。这样，任何带有返回值的命令均不能存储在列表中。其他的函数，如 glFlush()和 glFinish()不能存储在显示列表中的原因则在于它们依赖于客户的状态信息。还有，那些可能改变客户维护的状态值的命令也不能存储在显示列表中。

5.3.2 执行显示列表

在创建了显示列表之后，便可以通过调用 glCallList()来执行。同一显示列表可以多次执行，也可以与直接模式混合调用。

---

void glCallList (GLuint list);

该函数执行由 list 指定的显示列表。显示列表中的命令将按照它们的存储顺序执行。如果 list 未定义，则不执行任何动作。

由于能够改变 OpenGL 状态变量值的函数调用可以在显示列表中进行，所以执行显示列表时，这些值将被改变，就像在直接模式中调用这些命令一样。改变了的 OpenGL 状态值，在显示列表执行完毕之后仍然保持着。在例 5-2 中，在显示列表执行过程中改变的当前颜色和当前矩阵在调用结束之后依然有效，如例 5-3 所示。

例 5-3　显示列表执行之后，改变的状态值仍然有效

```
glNewList(listIndex,GL_COMPILE);
  glColor3f(1.0, 0.0, 0.0);
  glBegin(GL_POLYGON);
    glVertex2f(0.0,0.0)
    glVertex2f(1.0,0.0);
    glVertex2f(0.0,1.0);
  glEnd();
  glTranslatef(1.5,0.0,0.0);
glEndList();
```

尽管有时希望保持改变了的状态值，但有时却希望能在显示列表执行之前将状态变量值保存起来并在执行后恢复。使用 glPushAttrib()命令可将一组状态变量的值保存起来，而 glPopAttrib()命令则可在执行显示列表之后将状态变量恢复（关于这些命令的详细论述，请参见附录 2）。保存和恢复当前矩阵，可按“矩阵堆栈操作”中所描述的那样使用 glPushMatrix() 和 glPopMatrix()函数。为了恢复例 5-3 中的状态变量，可以使用如例 5-4 所示的代码。

例 5-4　在显示列表中恢复状态变量

```
glNewList(listIndex,GL_COMPILE);
  glPushMatrix();
  glPushAttrib(GL_CURRENT_BIT);
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_POLYGON);
      glVertex2f(0.0,0.0);
      glVertex2f(1.0,0.0);
      glVertex2f(0.0,1.0);
    glEnd();
  glTranslatef(1.5,0.0,0.0);
```

```
    glPopAttrib();
    glPopMatrix();
glEndList();
```

如果使用这种显示列表来恢复变量值，例 5-5 中的代码就能绘出一条绿色的不平移的直线。而在例 5-3 中，由于未进行数据的保存和恢复，所以绘制的线是红色的，并且产生平移。

例 5-5　使用显示列表恢复状态变量

```
void display(void)
{
    GLint i;
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 1.0, 0.0);
    for (i = 0; i < 10; i++)
      glCallList (listIndex);
    drawLine ();
    glFlush ();
}
```

只要其 OpenGL 内容表仍然为激活状态，glCallList()命令便可以在程序中任何位置调用。只要索引保持惟一确定，显示列表可以在一个程序中创建而在另一个程序中执行。另外，没有任何工具可将显示列表的内容存储到数据文件中，也没有任何工具可以由文件创建显示列表。在这一意义上来说，显示列表是为临时使用而设计的。当 OpenGL 内容表被破坏时，显示列表也同时被破坏。

5.3.3　显示列表的层次结构

创建的显示列表可以具有层次结构，即在一个显示列表中，通过在一对 glNewList()命令和 glEndList()命令之间调用 glCallList()，可以执行另一个显示列表。对于多组件组成的物体，尤其是其中某些部分需要多次使用时，分层显示列表是非常有用的。在下例中，便在用以渲染自行车的显示列表里，通过调用另一个显示列表来渲染自行车的某一部分。

```
glNewList(listIndex,GL_COMPILE);
    glCallList(handlebars);
    glCallList(frame);
    glTranslatef(1.0,0.0,0.0);
    glCallList(wheel);
    glTranslatef(3.0,0.0,0.0);
    glCallList(wheel);
glEndList();
```

为了避免无限的递归调用，显示列表的嵌套级别是受到限制的；嵌套级别一般是 64，也可能根据具体的 OpenGL 实现更多。调用 glGetIntegerv(GL_MAX_LIST_NESTING, GLint *data) 可以确定 OpenGL 实现对嵌套级别的限制。

OpenGL 允许一个显示列表调用另一个还未创建的显示列表。在后者还未创建时，调用为空操作。

通过分层显示列表，在一个列表外围加几个低层列表，可以近似一个可编辑的显示列表。如例 5-6 所示，将一个多边形置于显示列表中，多边形的顶点便可编辑。

例 5-6 使用分层显示列表

```
glNewList(1,GL_COMPILE);
glVertex3f(v1);
glEndList();
glNewList(2,GL_COMPILE);
glVertex3f(v2);
glEndList();
glNewList(3,GL_COMPILE);
glVertex3f(v3);
glEndList();
glNewList(4,GL_COMPILE);
  glBegin(GL_POLYGON);
    glCallList(1);
    glCallList(2);
    glCallList(3);
  glEnd();
glEndList();
```

调用显示列表 5 来渲染多边形。为了编辑顶点，仅需要重新创建一个对应于该顶点的单显示列表。由于一个索引值惟一确定地标识一个显示列表，用同一个索引创建显示列表时，将自动删除已存在的显示列表。

注意：尽管这项技术并不能保证优化使用内存或最佳的性能，但使用起来会带来许多方便。

5.4 显示列表及其索引的管理

到现在为止，我们都是用一个任意的整数来作为显示列表的索引。这在实际应用中是很危险的，因为用到了一个已经使用了的索引值，这样将覆盖已存在的显示列表。为了避免这种失误，可以使用 glGenLists()命令来产生一个未被使用的索引值并且用 glIsList()命令来确

定该索引是否被使用。另外，还可以使用 glDeleteLists()命令显式地删除某个指定的显示列表或一组显示列表。

GLuint glGenLists(GLsizei range);

该函数分配 range 个相邻的、未曾分配过的显示列表索引。返回的整数是一个索引值，该索引值标志着相邻的显示列表块的起始位置。如果返回的索引值全都标记为空而且被使用，在这些索引值被删除之前，对 glGenLists()的调用将不返回这些索引值。如果请求的索引值无效，或 range 值为零，则函数返回零。

GLboolean glIsList(GLuint list);

如果 list 已经被其他显示列表使用，返回 TRUE；反之，返回 FALSE。

在下面的例子中，请求一个索引值，如果成功，将用来创建一个新的显示列表：

```
listIndex=glGenLists(1);
if(listIndex!=0) {
  glNewList(listIndex,GL_COMPILE);
    ...
  glEndList();
}
```

命令 glDeleteLists()用来删除一组相邻的显示列表，并释放它们占用的索引值。

---

void glDeleteLists(GLuint list, GLsizei range);

删除显示列表 range，其起始点由 list 指定。如果企图删除一个并未创建的显示列表，将忽略该操作。

5.5　执行多个显示列表

OpenGL 提供了顺序执行多个显示列表的有效机制。该机制要求将显示列表索引值放入一个数组中，然后调用 glCallLists()命令。当一组显示列表的索引值对应着一系列有意义的值时，可以应用这种结构。例如，如果要创建一种字体，每一个显示列表索引值对应于该字体的一个字符的 ASCII 值。对有多种字体的情况，需要为每种字体建立一个不同的初始显示列表索引值。这些初始索引值可以通过调用 glCallLists()之前调用 glListBase()来指定。

---

void glListBase(GLuint base);

该函数指定一个偏移量，该偏移量被加到 glCallLists()中的显示列表索引中，以获取最终的显示列表索引值。缺省的显示列表基（base）是 0。列表基对只执行一个显示列表的 glCallList()或 glNewList()并没有影响。

---

void glCallLists(GLsizei n, GLenum type, const GLvoid *lists);

执行 n 个显示列表。所执行的列表索引值是由当前显示列表基（由 glListBase()指定）指明的偏移量加上由 lists 指向的数组中的有符号整数之和。

参数 type 指明数据类型和索引值数组的步长。通常是下列常量之一:

```
GL_BYTE
GL_UNSIGNED_BYTE
GL_SHORT
GL_UNSIGNED_SHORT
GL_INT
GL_UNSIGNED_INT
GL_FLOAT
```

也可以是 GL_2_BYTES, GL_3_BYTES 或 GL_4_BYTES，分别对应着 2，3 或 4 位的偏移量，而后利用下面的算法通过逐位相加来计算显示列表的偏移量。

```
offset = 0;
 for (i = 0; i < b; i++)
 offset = offset << 8;
 offset += byte[i];
 }
index = offset + listbase;
```

这意味着，对于多字节数据，由于各字节是顺次从数组中取出，最先得到的是高字节数据。

例 5-7 是从例 5-8 的完整程序中取出的一部分，该例说明了多显示列表的用法。该程序以 stroked 字体绘制字符（由一组线段构成的字符集）。子函数 initStrokedFont()为每一字母设置显示列表索引值，这样这些索引值就与字符的 ASCII 值相对应。

例 5-7　定义多显示列表

```
void initStrokedFont(void)
{
```

```
    GLuint base;
    base = glGenLists (128);
    glListBase(base);
    glNewList(base+'A', GL_COMPILE);
      drawLetter(Adata); glEndList();
    glNewList(base+'E', GL_COMPILE);
      drawLetter(Edata); glEndList();
    glNewList(base+'P', GL_COMPILE);
      drawLetter(Pdata); glEndList();
    glNewList(base+'R', GL_COMPILE);
      drawLetter(Rdata); glEndList();
    glNewList(base+'S', GL_COMPILE);
      drawLetter(Sdata); glEndList();
    glNewList(base+' ', GL_COMPILE);    /* space character */
      glTranslatef(8.0, 0.0, 0.0); glEndList();
}
```

glGenLists()命令分配了 128 个相邻的显示列表索引值。这些显示列表索引值的第一个值就是显示列表基。每一字母都需要创建一个显示列表；每一个显示列表的索引值是基值与该字符的 ASCII 值之和。在本例中，仅创建了几个字符和空格。

在显示列表创建之后，就可以调用 glCallLists()执行显示列表。例如，可以将一字符串传给 printStrokedString()函数。

```
void printStrokedString(GLbyte *s)
{
   GLint len = strlen(s);
   glCallLists(len, GL_BYTE, s);
}
```

在字符串中的每一字母的 ASCII 值作为显示列表索引值的偏移量进行传递。当前列表基与每个字符的 ASCII 值相加得到最后要执行的显示列表索引值。例 5-8 的输出结果如图 5-1 所示。

图 5-1　Stroked 字体中定义字符 A, E, P, R, S 的定义

例 5-8　利用多显示列表定义 Stroked 字体：stroke.c

```
#include <windows.h>

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL\glaux.h>

#define PT 1
#define STROKE 2
#define END 3

typedef struct charpoint
    GLfloat  x, y;
    int   type;
}   CP;

CP Adata[] = {  { 0, 0, PT}, {0, 9, PT}, {1, 10, PT},
              {4, 10, PT}, {5, 9, PT}, {5, 0, STROKE},
              {0, 5, PT}, {5, 5, END} };
CP Edata[] = {  {5, 0, PT}, {0, 0, PT}, {0, 10, PT},
              {5, 10, STROKE}, {0, 5, PT}, {4, 5, END} };
CP Pdata[] = {  {0, 0, PT}, {0, 10, PT},{4, 10, PT},
              {5, 9, PT}, {5, 6, PT}, {4, 5, PT}, {0, 5, END} };
CP Rdata[] = {   {0, 0, PT}, {0, 10, PT},  {4, 10, PT},
               {5, 9, PT}, {5, 6, PT}, {4, 5, PT},
               {0, 5, STROKE}, {3, 5, PT},{5, 0, END} };
CP Sdata[] = {   {0, 1, PT}, {1, 0, PT}, {4, 0, PT},
               {5, 1, PT}, {5, 4, PT}, {4, 5, PT},
               {1, 5, PT}, {0, 6, PT}, {0, 9, PT},
               {1, 10, PT},{4, 10, PT}, {5, 9, END} };

static void CALLBACK drawLetter(CP *l)
{
   glBegin(GL_LINE_STRIP);
   while (1) {
```

```
        switch (l->type) {
            case PT:
                glVertex2fv(&l->x);
                break;
            case STROKE:
                glVertex2fv(&l->x);
                glEnd();
                glBegin(GL_LINE_STRIP);
                break;
            case END:
                glVertex2fv(&l->x);
                glEnd();
                glTranslatef(8.0, 0.0, 0.0);
            return;
        }
    l++;
    }
}

void myinit (void)
{
    GLuint base;
    glShadeModel (GL_FLAT);
    base = glGenLists (128);
    glListBase(base);
    glNewList(base+'A', GL_COMPILE);
    drawLetter(Adata);
    glEndList();
    glNewList(base+'E', GL_COMPILE);
    drawLetter(Edata);
    glEndList();
    glNewList(base+'P', GL_COMPILE);
    drawLetter(Pdata);
    glEndList();
    glNewList(base+'R', GL_COMPILE);
```

```
    drawLetter(Rdata);
    glEndList();
    glNewList(base+'S', GL_COMPILE);
    drawLetter(Sdata);
    glEndList();
    glNewList(base+' ', GL_COMPILE);
    glTranslatef(8.0, 0.0, 0.0);
    glEndList(); }

    char *test1 = "A SPARE SERAPE APPEARS AS";
    char *test2 = "APES PREPARE RARE PEPPERS";
    void printStrokedString(char *s)
    GLsizei len = strlen(s);
    glCallLists(len, GL_BYTE, (GLbyte *)s);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix();
    glScalef(2.0, 2.0, 2.0);
    glTranslatef(10.0, 30.0, 0.0);
    printStrokedString(test1);
    glPopMatrix();
    glPushMatrix();
    glScalef(2.0, 2.0, 2.0);
    glTranslatef(10.0, 13.0, 0.0);
    printStrokedString(test2);
    glPopMatrix();
    glFlush();
}

int main(int argc, char** argv)
{
```

```
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 440, 120);
    auxInitWindow ("编程指南例 5-8");
   myinit ();
   auxMainLoop(display);
    return 0;
}
```

5.6　封装模式的改变

可以利用显示列表来组织和存储一组命令，以便在不同模式之间进行改变或设置各种参数。如果需要从一组设置切换到另一组时，利用显示列表比直接调用更为有效，因为这些设置是以一种与图形系统的要求相匹配的格式放在缓存中的。

例如，在多种纹理图之间切换时，显示列表可能更为有效（纹理映射将在第 10 章进行描述）。假设有两个不同的纹理，其大小恰好为纹理内存的容量。如果不用显示列表，就必须加载第一个纹理数据来画某些物体，然后等待将第二个纹理加载到内存，再利用该纹理绘制物体。若想再切换到第一个纹理时，还必须再一次将其调入内存而不是从纹理内存中直接取出。对于 OpenGL 来说，如果没有利用显示列表机制来提供一个“句柄”标识该内存，它就无法知道该纹理已经存储在内存中。使用显示列表，两个纹理就可以同时调入纹理内存，以后就可以随意使用而不必重新调入。

另外，在不同光照条件、光照模式以及材质参数设置之间进行切换时，显示列表比直接模式也更为高效（这些内容将在第 6 章进行详细讨论）。显示列表还可以用于点划模式、雾的参数以及裁剪平面方程中。通常情况下，显示列表的执行速度至少不会比相关的直接调用更慢，这一点是可以保证的。

注意：在使用显示列表时，会使系统产生一些额外的开销。

例 5-9 说明了如何利用显示列表在三种点划线模式中进行切换。首先，要调用 glGenLists() 来为每一点划模式分配一个显示列表，并为每种模式创建一个显示列表。这样，就可以调用 glCallList() 从一种点划模式切换到另一种模式之中。

例 5-9　利用显示列表进行模式变换

```
GLuint offset;
offset = glGenLists (3);
glNewList (offset, GL_COMPILE);
   glDisable (GL_LINE_STIPPLE);
glEndList ();
glNewList (offset+1, GL_COMPILE);
```

```
    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, 0x0F0F);
glEndList ();
glNewList (offset+2, GL_COMPILE);
    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, 0x1111);
glEndList ();
#define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES); \
    glVertex2f ((x1),(y1)); glVertex2f ((x2),(y2))
glEnd();
glCallList (offset);
drawOneLine (50.0, 125.0, 350.0, 125.0);
glCallList (offset+1);
drawOneLine (50.0, 100.0, 350.0, 100.0);
glCallList (offset+2);
drawOneLine (50.0, 75.0, 350.0, 75.0);
```

第 6 章 颜色

本章主题

绝大部分 OpenGL 应用程序的目的是在屏幕窗口中绘制彩色图形。窗口是像素点的矩形阵列，每一个像素点都包含并显示其自身带有的颜色。因此，从某种意义上讲，OpenGL 的实现所执行的各种计算——综合考虑 OpenGL 命令、状态信息和各个参数的数值的计算——其目标是确定最终画在窗口里的每一个像素点的颜色。本章将详细说明用于指定颜色的函数，以及 OpenGL 如何理解这些函数，共分为下面几个部分：

- "颜色的感知"中讨论人的眼睛是如何感知颜色的。
- "计算机的颜色"中阐述了计算机显示器上的像素点与其颜色之间的关系，同时定义了两种显示模式：RGBA 方式和颜色索引模式。
- "RGBA 模式和颜色索引模式"中论述了这两种显示方式是如何操作图形硬件，以及如何确定应使用哪一种显示方式的。
- "指定颜色和阴影模式"中详细说明了用于指定需要的颜色和阴影模式的 OpenGL 命令。

6.1 颜色的感知

按照物理学的观点，光是由微小的光子微粒组成，每一个光子微粒都沿各自的轨迹运动，并以各自的频率（或者波长，或者频率与波长的任何一种能量形式，或者由能量确定的其他形式）振动。位置、运动方向和频率/波长/能量描述出了光子所有的特性。光子的波长从约 390nm（紫色）到 720nm（红色），包括了可见光谱中的所有颜色，形成一个五彩缤纷的彩色带（紫色、靛青、蓝色、绿色、黄色、橙色和红色）。

当然，人的眼睛能够感知的颜色并不一定是上述色带中的颜色，比如白色、黑色、褐色和粉红色等。其中的奥妙在于，眼睛实际看到的颜色是不同频率的光混合后的结果。实际光源发射的光子频率分布描述了光的特性。各种频率的光，经等量混合后将得到理想的白光。激光通常是单色光，其中的全部光子具有同样的频率，相同的方向和相位。钠灯光的主要成分是黄色光。宇宙的星球发出的光的频率分布，很大程度上依赖于它们自身的温度（黑体辐射）。在我们周围的环境中的大多数光源，发出的光的频率分布就要更复杂一些。

下面我们简要论述一下人眼视觉的产生。当眼睛视网膜里的特定细胞（称为锥体细胞）受到光的刺激而兴奋时，产生神经冲动穿入大脑，人便看到了颜色。眼结构中三种不同的锥

体细胞将对三种不同波长的光表现出最敏感的反应：这三种光分别是红光、绿光和蓝光（患有色盲症的人通常是缺少一种或多种锥体细胞）。当混合光进入人眼，视网膜上的不同感光锥体细胞将记录下各自的兴奋程度。如果另一束进入眼睛的混合光引发的三种锥体细胞的兴奋度与前面的光相同，眼睛将无法区分该混合光与前面的混合光有何区别。

正是由于眼睛把各种颜色记录为锥体细胞接受光刺激时产生的兴奋程度，因此，眼睛可以感受到棱镜或彩虹光谱以外的颜色。举个例子来说，若发射红蓝混合光，视网膜上感应红色和蓝色的锥体细胞将同时兴奋，这时，眼睛所看到的光是洋红色的（不在光谱中）。以其他形式混合各种光将产生褐色、青绿色和紫红色等不在光谱中的光照效果。

计算机的显示器也是通过在发光像素点上按一定比例混合红、绿和蓝三色来模拟可见颜色。所谓一定比例，就是使得发光像素点刺激眼睛视网膜上的红、绿和蓝感觉锥体细胞所达到的兴奋级别，与其所模拟的实际光刺激所产生的兴奋级别相同。显示器中的三种颜色光射线管一一对应着感光锥体细胞。如果人类有第四种锥体细胞，如黄色感觉细胞，那么彩色显示器就需要同时包括一个发射黄光的射线管，而我们将采用包括四个分量的 RGBY（红色、绿色、蓝色和黄色）来定义颜色。反之，如果人人都是同一类型的色盲，那么这里讨论的内容就会变得更加简单。

为了显示一种特定的颜色，显示器发射恰当数量的红光、绿光和蓝光以刺激眼睛中不同类型的锥体细胞。一个彩色显示器可以向各个像素点发射不同比例的红光、绿光和蓝光，这样眼睛就能看到成千上万个带有各自颜色的微小光点。

这节内容仅仅考虑到眼睛是如何感知混合光的。光从物体表面反射回来进入眼中的情形要复杂得多，例如，白光照射到一个红球上，反射回来的是红光，照射在蓝色玻璃上的黄光，在玻璃的另一面观察的将是一片漆黑。在“真实世界与 OpenGL 中的光照”这一部分中将详细论述这些现象。

6.2 计算机的颜色

在一个彩色的计算机屏幕上，硬件激发屏幕上的每一个像素点发出不同数量的红光、绿光和蓝光，这些光的数量称为数值 R、G 和 B。这三个值通常是放在一起（有时还包括第四个值，称为 alpha 或 A），统称为 RGB（或 RGBA）。每一个像素点的颜色信息既可以以 RGBA 方式存储，也可以以颜色索引的方式存储。RGBA 模式中的 R、G、B（和有时会存在的 A 值）三个数值对应于每一个像素点，而在颜色索引模式中，对应于每个像素点的是单个数值（称为颜色索引）。每个颜色索引表是一个定义了 R、G 和 B 值的特定集合，其中的一个条目即是一个颜色图。

在颜色索引模式下，有时需要改变颜色图中的数值。然而，由于颜色图是由窗口系统控制的，所以任何 OpenGL 命令都无法完成此操作。本书中所有的例程在初始化颜色显示模式的同时，通过调用辅助库里的程序打开窗口，这一点在附录 5 中进行了详细的说明。

在像素点阵列的大小和可以显示在每个像素点上的颜色数目这两个方面不同的图形硬件之间有很大的区别。在一个给定的图形系统中，每一个像素点都提供了相同数量的内存空间以存储颜色值，而所有像素点内存空间的总和就是所谓的颜色缓存。缓存的大小一般以字节数来衡量，8 个字节的缓存可以存储 8 个字节的像素点数据（256 种不同的颜色）。不过，不同计算机的缓存大小不尽相同。

R、G 和 B 在 0.0（强度最小）和 1.0（强度最大）之间取值，例如，R = 0.0, G = 0.0, 和 B = 1.0 代表最亮的蓝色；如果 R、G 和 B 都是 0.0，该像素点为黑色；而如果 R、G 和 B 都是 1.0 时，为最亮的白色。绿色和蓝色混合将生成青色，而蓝色和红色混合后得到的是洋红色。为了帮助读者学习如何使用 R、G 和 B 以生成所需要的颜色，见彩图 12 和13所示的彩色立方体。该立方体的轴线分别代表红色、蓝色和绿色的强度。图 6-1 显示的是一个黑白立方体。

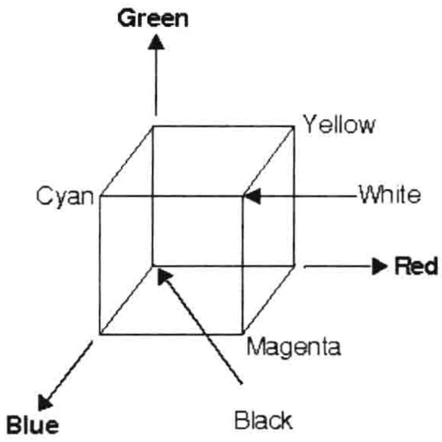


图 6-1　变换成黑白图像的彩色立方体

可以使用如下简单的命令行来指定一个物体（此处为一点）的颜色：

```
glColor3f (1.0, 0.0, 0.0);
/* 当前的 RGB 颜色为红色：纯红色，无绿色和蓝色成分 */
glBegin (GL_POINTS);
glVertex3fv (point_array);
glEnd ();
```

某些模式下（例如执行光照计算或纹理映射计算）时，在将颜色作为体现像素的颜色值存入帧缓存之前，分配的颜色值在各种其他操作中将一直有效。事实上，该像素的颜色是经由一系列的操作后才最终确定的。

在程序执行之初，必须先设置颜色的显示方式，要么是 RGBA 模式，要么是颜色索引方式。颜色的显示方式一旦确定下来，就不能更改。在程序执行的过程中，每一个几何图元的颜色值（一个颜色索引或一个 RGBA 值）都是根据各个顶点要素来确定的。无论使用由自己明确指定的顶点颜色值，还是激活了光照，转移矩阵、物体表面法线和其他一些材料特

性都将相互作用，最终决定顶点的颜色。换句话说，一个红色的球在蓝色光的照射下与没有光照时就是不一样。在执行完相应的光照计算后将执行选择的阴影计算。正如在指定颜色和阴影模型这节内容中所阐明的那样，用户可以选择没有明暗差别的阴影或平滑的阴影，选择不同模型将导致像素最终颜色效果有所不同。

然后对图元进行光栅化，即变换成一个二维图像。光栅化包括确定该图元占用了窗口坐标中一个整型栅格里的哪些正方形区域，和在此之后为每个这样的正方形分配的颜色和其他数值。一个栅格状正方形加上与其相关的颜色值、深度值（z）和纹理坐标就称为一个片原。像素点是帧缓存中的元素；一个片原来源于一个图元，同时结合与之相对应的像素点以产生一个新的像素点。片原结构一旦建立，就可以对片原进行纹理映射、加雾和反模糊化计算（如果这些操作处在激活状态）。在此之后，任何一个特定的 alpha 混合、抖动和二进制逻辑操作都通过已经存储在帧缓存里的片原和像素点进行操作。最后，将片原的颜色值（颜色索引或者是 RGBA）赋给该像素点，并且以窗口的颜色显示模式将该颜色显示在窗口中。

6.3 RGBA 模式与颜色索引模式

6.3.1 RGBA 模式与颜色索引模式的对比

无论是在颜色索引模式还是在 RGBA 模式中，每个像素点都存储着一定量的颜色数据。帧缓存的位平面数目确定了这一存储量。一个位平面为一个像素点容纳一位数据。如果有 8 个位平面，那么，每一个像素点就有 8 个字节的颜色数据，因此，每一个像素点就可以有 2^8=256 种不同的颜色。

通常把位平面均匀地划分为 R、G 和 B 三个存储空间（也就是说，一个 24 位的位平面系统为红色、绿色和蓝色分别分配 8 个字节的存储空间），但并不能一概而论。利用带有参数 GL_RED_BITS,GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS 或 GL_INDEX_BITS 等的函数 glGetIntegerv()，可以获得系统分配给红色、绿色、蓝色、alpha 或者颜色索引值的可用位平面数目。

注意：对于大部分计算机的屏幕，人眼感受到的颜色强度并不是按线性分布的。以只有红色分量，而绿色和蓝色值为零的颜色为例，当颜色强度值从 0.0（最小）到 1.0（最大）变化时，打在屏幕像素点的电子数目将不断增加。那么，强度值为 0.5 时，像素点显示的颜色是否看起来是强度值分别为 0.0 和 1.0 时的两种颜色强度的平均呢？为了验证这一点，我们编写一个应用程序，以西洋跳棋盘图案方式交替绘制强度值为 0.0 和 1.0 的像素点，同时与颜色强度值固定为 0.5 所画区域相比较。OpenGL 默认这两个图案的颜色是相同的。如果不相同，程序员应该使用系统所提供的某种修改机制进行调整。例如，许多系统都包含一个用于调整颜色强度的工作表，通过该表进行调整使得强度值为 0.5 时显示的颜色看起来是强度值分别为 0.0 和 1.0 时的两种颜色的平均。通常使用的映射方式是一种指数形式（指数为

gamma），即 gamma 修正。对红色、绿色和蓝色三个分量使用相同的 gamma 值进行修正可以得到很好的效果，采用三个不同的 gamma 值会给出更好一点的效果。

6.3.2　RGBA 显示模式

在 RGBA 模式中，硬件给各个 R、G、B、A 分量设置了一定数量的位平面（每一分量的位平面的数量并不一定相同），如图 6-2 所示。一般地，R、G 和 B 的数值以整数形式而不是以浮点数形式进行存储，而且，对于这些数值的存取，都是按一定比例将它们变换成可用字节数的大小来实现的。举个例子来说，如果系统给予 R 分量 8 个可用字节，可以表示 0 到 256 之间的所有整数，那么，在位平面里的 0, 1, 2, ..., 255 对应的 R 值是 0/255 = 0.0, 1/255, 2/255, ...,255/255 = 1.0。不考虑位平面的个数，0.0 表示最小的强度，而 1.0 表示最大的强度。

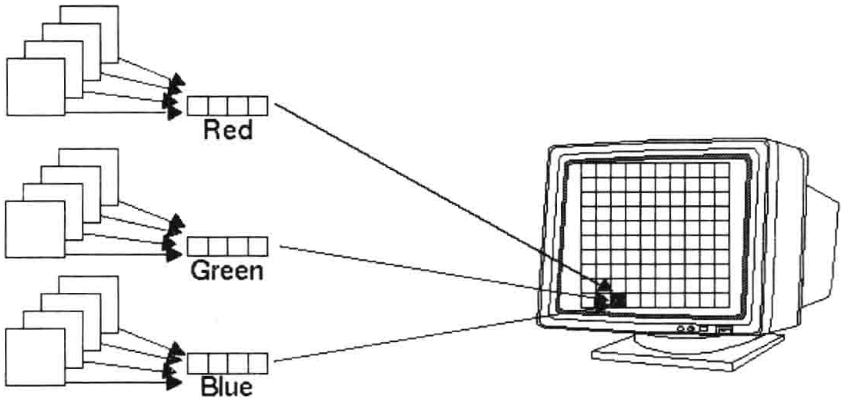


图 6-2　由位平面产生的 RGB 值

注意： alpha 值（即 RGBA 中的 A 值）对屏幕上显示的颜色没有直接影响。但 alpha 值可以用于颜色混合和确定透明度等许多操作上，而且对 R、G 和 B 值都有一定的影响。关于 alpha 值的详细内容可参阅颜色的混合操作这节。

在单个像素点上可以显示的不同颜色数量的多少，依赖于位平面的数量和用于解释那些位平面的硬件容量。不同颜色的数量不能超过 2^n，这里 n 是位平面的个数。那么，拥有 24 位位平面的显示器最多可以显示 1677 万种不同颜色。

6.3.3　抖动操作

某些图形硬件运用抖动操作来增加可显示颜色的数目，但这是以降低空间分辨率为代价的，抖动操作是一种将几种颜色结合起来以产生其他颜色效果的技术。为了说明抖动操作是如何工作的，首先假设系统只给 R、G 和 B 各自分配一个字节，那样的话，这个系统只能显示八种颜色：黑色、白色、红色、蓝色、绿色、黄色、蓝绿色和洋红色。为了显示一个粉红色的区域，硬件可以以跳棋盘的方式（即红色像素点和白色像素点交替分布）填充该区域，如果眼睛离屏幕远到一定程度（不能够分辨出单个像素点）时，该区域便表现

为粉红色（一种介于红色和白色之间的颜色）。如果填充的红色像素点所占的比例大，该区域的颜色是一种偏红的粉红色；而如果填充的白色像素点所占的比例大时，其颜色就是一种偏白的粉红色。

在没有粉红色的像素点情况下，得到“粉红色”效果的惟一方法就是运用这一技术来覆盖许多像素点组成的区域——当然，单个像素点不能进行抖动操作。如果将 RGB 值定义为一种系统无法得到的颜色，然后用该颜色填充多边形，实际上硬件将用与所需颜色相近的颜色混合物填充该多边形内部的像素点。虽然程序员可以从帧缓存里读出像素点的信息，但是，由于根本就不存在粉红色，得到的是实际上是红色和白色像素点的颜色值。关于读取像素点信息这方面的内容请参阅第 9 章。

图 6-3 例示了一些将黑色和白色进行简单的抖动操作以产生灰色阴影的例子。从左到右，图顶端的三个 4×4 方格图案分别表示 50%、19%和 69%灰色度的抖动操作结果。对应每个图案的下面，用该图案取代图案中的黑色和白色区域，如此循环一定的次数，得到图中底端的三个图案，但是，那些黑色和白色方形区域依然比绝大多数像素点的尺寸大。如果距离足够远——比如在房间的另一端观察，便会发现黑色和白色混杂在一起，看起来才是三个不同程度的灰色图案。

若 R、G 和 B 各有 8 个字节，那么不必通过抖动操作便可以得到相当高品质的图像。这仅仅因为显示器是 24 位颜色的位平面，并不意味着抖动操作不会发生。例如，如果计算机是在双缓存模式下运行，有可能将其位平面分成两组，每组各 12 个位平面，这样一来，实际上 R、G 和 B 这三个分量各自只有 4 个字节。要是不进行抖动操作，4 字节的颜色分量在很多情况下不能给出令人满意的效果。

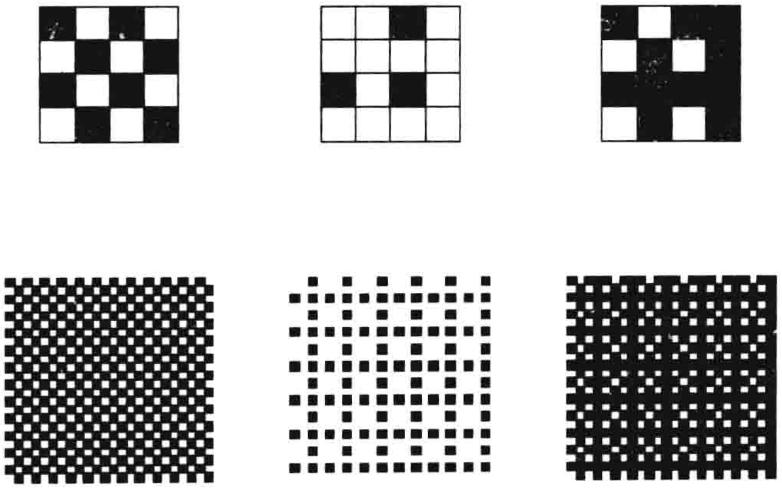


图 6-3　用黑色和白色进行抖动操作以产生灰色效果

通过使用参数 GL_DITHER 调用函数 glEnable() 或 glDisable()，可以激活抖动操作或使该操作失效。

6.3.4　颜色索引显示模式

在颜色索引模式下，OpenGL 使用一个颜色图（或者说一个查询表）来确定颜色。这相当于使用一个调色板，通过混合各种颜料来绘制一个场景，这些颜料按数字顺序排列。通常，画家的调色板都留有用于混合颜料的地方，同样，一个计算机的颜色图将提供一些索引，其中基本的红色、绿色和蓝色的值也可以相互混合，如图 6-4 所示。

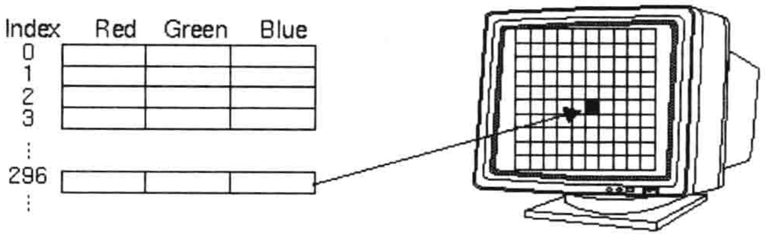


图 6-4　颜色图

为了获得一个按数字顺序排列的颜料，画家从调色板中选择一种颜色并将该颜色填充到相应数字的区域中，而计算机在位平面里存储每个像素点的颜色索引。然后参考颜色图来确定位平面的数值，并用颜色图中相应的红色、绿色和蓝色值来画屏幕，如图 6-5 所示。

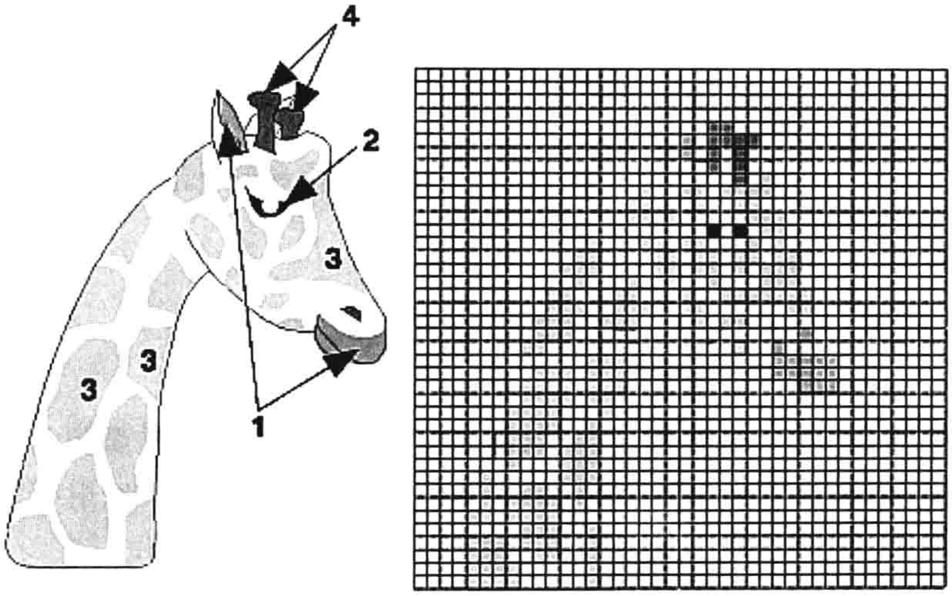


图 6-5　使用颜色图绘制图形

在颜色索引模式下，同时能够获得的颜色数目受颜色图的大小和可用位平面数目的限制。颜色图的大小取决于硬件（颜色图专用）的数量，通常是在 256（28 位）和 4096（212 位）的范围内。在颜色索引模式中，颜色图大小为 2 的 n 次幂（n 为有限整数），由可用位平面数目来编辑索引。如果在颜色图中有 2^n 个索引，而且有 m 个可用位平面，则可用的索引数目的大小为 2^n 和 2^m 两者间的较小值。

在 RGBA 模式中，各个像素点的颜色值是相互独立的。而颜色索引模式中，使用位平面中相同索引的像素点将共享颜色图中的同一个位置。当颜色图中某个索引的内容发生改变时，对应该颜色索引的所有像素点的颜色也随之发生改变。

6.3.5 在 RGBA 模式与颜色索引模式之间进行选择

在选用 RGBA 模式还是颜色索引模式这一问题上，可以根据可用硬件的种类和应用程序所需决定。对于大多数系统而言，RGBA 模式可以同时描述较多的颜色，而颜色索引模式下则较少。而且，对于一些颜色效果，诸如阴影模型、光照模型、纹理映射模型和雾模型等等，RGBA 模式提供了比颜色索引模式更大的灵活性。

在下列情况下，更适合使用颜色索引模式。

- 如果载入一个已经使用了颜色索引模式的应用程序，为下面的工作方便起见，最好不要将该模式变换成 RGBA 模式。
- 如果只能获得很少的 n 个可用位平面数目，同时并不需要超过 2^n 种不同的颜色，应该考虑使用颜色索引模式。
- 如果可用位平面的数量有限，RGBA 模式可能产生很明显的粗糙阴影。要是对阴影模型的要求比较苛刻，颜色索引模式将在这一方面带来令人满意的效果（如灰色阴影）。
- 对于许多绘画技巧，如颜色图动画和层图等，颜色索引模式是非常有效的。这一类的详细内容请参阅第 13 章。

通常，纹理映射、光照、阴影、雾、锯齿现象的消除和颜色混合等操作在 RGBA 模式下的工作效果更好。

6.4 指定颜色和阴影模型

OpenGL 保持着一个当前颜色（在 RGBA 模式下）和一个当前颜色索引（在颜色索引模式下）。除非程序员使用一个更复杂的颜色模型如光照模型或纹理映射模型，否则，绘制的任一个物体是使用当前颜色（或当前颜色索引）。请看以下伪代码：

```
set_color(RED);
draw_item(A);
draw_item(B);
set_color(GREEN);
set_color(BLUE);
draw_item(C);
```

A 和 B 两项为红色，而 C 项为蓝色。将当前颜色设定为绿色的第四个函数是无效的（并且浪费了时间）。如果不存在光照模型或纹理映射模型，一旦设定当前颜色，就用该颜色绘

制随后的所有项，直到当前颜色发生改变为止。

6.4.1 在 RGBA 模式下指定颜色

在 RGBA 模式下，可以使用 glColor*()函数来选择当前颜色。

---

void glColor3{b s i f d ub us ui} (TYPEr, TYPEg, TYPEb);
void glColor4{b s i f d ub us ui} (TYPEr, TYPEg, TYPEb, TYPEa);
void glColor3{b s i f d ub us ui}v (const TYPE*v);
void glColor4{b s i f d ub us ui}v (const TYPE*v);

这一组命令设置当前的红色、绿色、蓝色和 alpha 值。命令至多有三个后缀，用于区分参数的变化。第一个后缀为 3 或 4，说明是否除红色、绿色和蓝色值之外又加上了一个 alpha 值。如果没有加 alpha 值，系统将把它自动设置为 1.0。第二个后缀说明参数的数据类型：字节型、短整型、整型、浮点型、双精度型、无符号字节型、无符号短整型、或者是无符号整型。第三个后缀 v 是可选择的，说明命令参数为一指向给定类型的数组的指针。

对于参数为浮点型数据的函数 glColor*()，其参数在 0.0 和 1.0（存储在帧缓存内的最小值和最大值）之间取值（区间[0,1]之外的值，在直接使用时将截取到区间[0,1]之内，但是用于修改光照材料参数时该值不变）。无符号整型颜色分量值一旦指定，将被线性映射为浮点型数值，其中将能够描述的最大整型值映射为 1.0（最大强度），而零映射为 0.0（零强度）。有符号整型颜色分量值线性映射为浮点型数值时，其正的最大可描述整型值映射为 1.0（最大强度），负的最大可描述的整型值映射为-1.0（如表 6-1 所示）。

表 6-1 将颜色值变换成浮点数

后缀	数据类型	最小颜色值	最小映射值	最大颜色值	最大映射值
b	1 字节整数	-128	1.0	127	1.0
s	2 字节整数	-32,768	-1.0	32,767	1.0
i	4 字节整数	-2,147,483,648	-1.0	2,147,483,647	1.0
ub	无符号 1 字节整数	0	0.0	255	1.0
us	无符号 2 字节整数	0	0.0	65,535	1.0
ui	无符号 4 字节整数	0	0.0	4,294,967,295	1.0

浮点型数据则进行直接映射。在更新当前颜色之前，无论是浮点数还是有符号整数都不会被截取在区间[0,1]内。然而，当对各颜色分量进行插值或将它们写入颜色缓存之前，要将这些分量值缩放到区间[0,1]内。

6.4.2 在指定颜色索引模式下指定颜色

在颜色索引模式下，常常调用 glIndex*()函数，选择一个单值的颜色索引作为当前颜色索引。

---

void glIndex{sifd}(TYPE c);
void glIndex{sifd}v(const TYPE *c);

这一组函数设置当前的颜色索引值，第一个后缀说明参数的数据类型：可为短整型、整型、浮点型或双精度型。第二个后缀 v 是可选的，指出参数是给定数据类型下颜色值队列的一个指针（该队列中只包含一个数值）。

当前的索引值按浮点类型存储。整数不必按特殊的映射方式，而是直接变换成浮点数。颜色索引缓存的可描述范围之外的索引值不作截取处理。然而，在一个索引进行了抖动操作（当抖动操作被激活）并且将写入帧缓存之前，该索引将变换成定点的格式。导出的固定点数值的整数位中，不对应帧缓存上的位的部分都将被消隐掉。

6.4.3 指定阴影模型

一个线段图元或一个填充的多边形图元既可以用单一的颜色（无明暗差别的阴影）也可以用许多不同的颜色（光滑阴影，也称为 Gouraud 阴影）来绘制。程序员可以用函数 glShadeModel()来指定期望的阴影模型。

---

void glShadeModel (GLenum mode);

使用无明暗差别的阴影时，图元某个顶点的颜色将重复用在该图元的所有顶点上。而在使用光滑阴影时，图元各个顶点的颜色是相互独立的。对于线段图元，线段上的平滑颜色是在两顶点颜色之间插值得到的。对于多边形图元，多边形内部区域的颜色也是在相邻顶点颜色间插值得到的。例 6-1 将绘制一个使用光滑阴影的三角形，如彩图 11 所示。

例 6-1 画一个使用光滑阴影模型的三角形：smooth.c

```
#include "windows.h"


#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>


void myinit (void)
```

```
{
   glShadeModel (GL_SMOOTH);   /* GL_SMOOTH 为缺省状态 */
}

void triangle(void)
{
   glBegin (GL_TRIANGLES);
   glColor3f (1.0, 0.0, 0.0);
   glVertex2f (6.0, 6.0);
   glColor3f (0.0, 1.0, 0.0);
   glVertex2f (26.0, 6.0);
   glColor3f (0.0, 0.0, 1.0);
   glVertex2f (6.0, 26.0);
   glEnd ();
}

void CALLBACK display(void)
{
   glClear (GL_COLOR_BUFFER_BIT);
   triangle ();
   glFlush ();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
 {
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
     gluOrtho2D (0.0, 30.0, 0.0,
                 30.0 * (GLfloat) h/(GLfloat) w);
   else
     gluOrtho2D (0.0, 30.0 * (GLfloat) w/(GLfloat) h, 0.0,
         30.0);
   glMatrixMode(GL_MODELVIEW);
```

```
 }

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 6-1");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return 0;
}
```

使用光滑阴影时，相邻的像素点颜色值之间有轻微的不同。在 RGBA 模式下，相邻像素点的颜色有轻微差别看起来也是很相似的，因此多边形区域内的颜色变化是渐进的。在颜色索引模式下，相邻像素点的颜色索引值对应颜色索引表中的不同位置，其颜色可能根本不一样。相邻的颜色索引条目有可能包含范围相差悬殊的不同颜色，因此，在颜色索引模式下以光滑阴影模型绘制的多边形看起来给人一种光怪陆离的感觉。

为了避免这个问题，程序员可以自己创建一个颜色值按条目顺序连续变化的颜色图。记住，在这种情况下，应该通过窗口系统而不是 OpenGL 来将设置的颜色值载入颜色图中。当然，就目前而言，可以使用函数 auxSetOneColor()来载入颜色图中的一个指定了红、绿、蓝颜色值的索引。该函数的第一个参数是索引号，其他参数为红、绿和蓝颜色值。为了载入 32 个连续的稍有不同黄色阴影的颜色索引号（从 16 号到 47 号），可以这样来调用：

```
for(i=0; i<32; i++)
{
   auxSetOneColor(16+i, 1.0*(i/32.0), 1.0*(i/32.0), 0.0);
}
```

现在，如果仅仅使用索引号为 16 到 47 的颜色渲染使用了光滑阴影的多边形，多边形将显现出逐渐变化的黄色阴影。而使用无明暗差别的阴影，单个顶点的颜色就确定了整个图元的颜色。对于一条线段，当第二个点（终点）已经确定，该线段的颜色就设置为当前的颜色。对于一个多边形，当某一特定的顶点确定之后，其填充的颜色也是当前起作用的颜色，如表 6-2 所示。该表从 1 开始计算顶点和多边形的数目。虽然 OpenGL 严格遵守这些规则，但是，在绘制采用了无明暗差别的阴影的图元时，为了避免产生不确定性，最好的办法是仅给该图元指定一种颜色。

表 6-2　为第 i 个使用无明暗差别的阴影的多边形选择颜色

多边形类型	用于给第 i 个多边形选择颜色的顶点
单多边形	1
三角形带	i+2
三角形扇面	i+2
不相连三角形	3i
方形带	2i+2
不相连方形	4i

第 7 章　光照

本章主题

在前面已经提到，为了显示最终的场景，OpenGL 需要计算出存放在帧缓存中每个像素点的颜色。一些计算依赖于场景中的光照，以及场景中的物体对光线的反射或吸收。这一点是贴近实际生活的：在晴空万里与乌云密布时，大海的颜色完全不同。阳光普照时海水看起来是亮蓝色的，乌云密布时则是灰绿色的。在具体操作中读者会发现，对于大多数物体，只有加入了光照条件后才会有较鲜明的三维效果。图 7-1 的左右两侧所显示的球体是完全相同的，只是右侧没有光照。

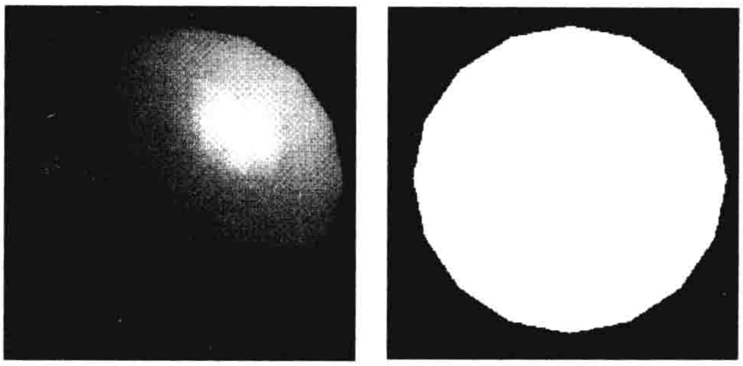


图 7-1　被照亮和未被照亮的球体

正如图中所示的那样，未被照亮的球体和一个二维的圆盘比起来几乎没有什么区别。可见，在创建三维场景时，物体和光线的相互作用是何等的重要。

在 OpenGL 中，通过对场景中的光线和物体进行不同操作，可以产生各种不同效果。本章解释了如何控制场景中的光照，同时还讨论了 OpenGL 中光照模型的概念，并详细描述了如何设置各种光照参数以实现某种特定效果。最后，本章给出了用于确定光照对物体颜色的影响的光照计算。本章的主要内容安排如下：

- “真实世界与 OpenGL 光照”，深入浅出地解释了真实世界中的光照现象，并阐述了 OpenGL 中如何模拟实现这一过程。
- “一个简单的例子：光照球体的渲染”，通过引入一个渲染光照球体的简单程序，介绍了 OpenGL 的光照工具。
- “创建光源”，解释了怎样定义光源并确定光源位置。
- “选择光照模型”，讨论了光照模型的基本元素及其设定。
- “材料属性的定义”，解释了如何定义物体的材质属性，使得物体和光线按预期的效

果相互作用。

- “光照的数学计算”，给出了 OpenGL 用来确定场景中的光照效果的数学计算。
- “颜色索引模式下的光照”，论述了 RGBA 模式与颜色索引模式下光照的区别。

7.1　真实世界与 OpenGL 光照

在观察物体的表面时，人眼睛对颜色的感知取决于进入眼内视锥细胞的光子的能量分布。这些光子来自于光源或发光物体，一些被表面吸收，而另一些则被反射。而且，不同表面的反射特性的差别是非常大的：对于较光滑的表面，入射光线在某个方向上将完全反射，在另外的方向上则完全散射。大部分表面处于全反射与散射之间。

OpenGL 近似地将光线分解成红、绿、蓝三种成分。光源的颜色取决于它所发出的红、绿、蓝光的和。表面的材质特征则是由这个表面在各个方向反射的红、绿、蓝光的百分比决定。虽然 OpenGL 的光照方程仅是一种近似算法，但与实际是十分接近的，并且可以达到很快的计算速度。如果对精确性提出了更高的要求，程序员就必须自己编写进行计算的程序代码。要知道，这类软件的设计是非常复杂的，在编程知识的基础上还需要大量的光学知识。

在 OpenGL 的光照模型中，场景中的光线来自不同光源，这些光源的开关相互独立，有的从特定的方向、位置射出，还有一些在场景中散射。例如，扭亮房间中的电灯时，进入眼内的光线大部分直接来自灯泡，还有一部分光线是经过了一面、两面、或几面墙的反射之后进入观察者眼中的。经过反射后的光线（称作环境光）无法确定其最初来自哪个方向。但是当关闭某个特定光源后，某些反射光就会消失。

场景中还有另外的环境光，这些光经过了多次散射，甚至已经无法确定其最初光源在何处。

OpenGL 的光照模型假定光源只对吸收或反射光线的表面产生影响，同时认为每个表面都可以由不同属性的材质构成，材质本身也可能产生光线（如汽车的前灯），并能将入射光线散射到各个方向，或者像镜子等其他光滑表面那样将大部分光线反射到特定方向。

OpenGL 的光照模型将光照分成四个独立的成分：发射光，环境光，散射光，镜面反射光。四种成分独立地计算，然后叠加到一起。

7.1.1　发射光，环境光，散射光和镜面反射光

发射光是最简单的，它由物体发出，并且不受任何光源的影响。

背景光成分是从某个光源发出并由环境经多次散射得到的，难以确定其最初的方向，看起来好像来自于四面八方。一个房间中的逆光便含有大量的背景光成分，因为此时到达人眼中的大部分光线已经过了多次反射。而户外的聚光灯光源场景中，背景光成分很少：大部分光线都是直线传播，而且因为是在开阔的户外，很少有光线在经过反射后才进入眼中。背景光照射到一个表面上时，将均匀地在各个方向都产生散射。

散射光来自于一个固定方向，所以当光线垂直照射到物体表面上时要比斜照时亮一些。然而，一旦光线照到表面上就均匀地在各个方向都发生散射，因此不论观察点处在哪个位置，其亮度都是一样的。来自于特定位置与方向的任何光线都可能带有散射光成分。

镜面反射光也来自于特定方向，但它几乎全部反射到对称的方向上。一束校准好的激光射到高质量的镜面上几乎可以产生 100%的镜面反射。光滑的金属或塑料也可以产生较高的镜面反射成分，而白灰墙或地毯几乎不发生镜面反射。镜面反射度可以看成是代表着物体的光洁度。

尽管光源发出的光是按单一频率分布的，但其中的环境光、散射光和镜面反射光分量是有区别的。例如，一束白光照射到房间中的红色墙壁上，尽管入射光是白色的，但散射光变成了红色。

注意： OpenGL 允许独立地为光线的各个分量设置红、绿、蓝值。

7.1.2 材料颜色

OpenGL 的光照模型进行了这样的近似处理：材料的颜色，是由它对红、绿、蓝光各反射百分比的叠加结果决定的。例如，纯红色的球体将反射全部的红色入射光，并吸收全部的绿色、蓝色入射光。在白光（由等量的红、绿、蓝光组成）中来看这个球体时，由于反射红光，球体将显现为红色的。如果把球放在纯红色的光中，球体自然也是红色的。但是，若在纯绿色的光中观察，由于绿色被吸收，所以没有反射光，球体看起来将会是黑色的。

同光线一样，材料也有不同的背景色、散射色和镜面反射色，它们决定了材质对背景光、散射光和镜面反射光的反射系数。一种材质的环境光反射系数是每种入射光中环境颜色成分的叠加，同样道理，散射光反射系数是对入射光中散射颜色成分的叠加，镜面反射光的反射系数亦然。环境光和散射光的反射系数决定了材质的颜色。镜面反射光的反射系数通常是白或灰，其亮度与光源中镜面反射光的强度有关。如果考虑一下白光照在光滑的红色塑料球体上的情况，大部分球体是红色的，但正对入射光的高亮度区是白色的。

7.1.3 光线与材质的 RGB 值

为光线指定颜色成分意味着光的颜色与材料的颜色是不同的。对光线来说，每种颜色的数值对应于它所占光强的百分比。如果光线的 R, G 和 B 值都是 1.0，这种光就是最亮的白光。如果三个值都是 0.5，颜色仍然是白色，但亮度只有原来的一半，所以这种光看起来是灰色的。如果 R=G=1 而 B=0（全红和全绿，没有蓝），此时光是黄色的。

对材质来说，其 R、G、B 值对应着材质对这些颜色光的反射比例。所以，若某材料 R=1, G=0.5, B=0，说明这种材质反射全部红光，一半的绿光，不反射蓝光。换句话说，在 OpenGL 中，若光线的 RGB 值为(LR, LG, LB)，材料的 RGB 值为(MR, MG, MB)，如果其他的反射影响可以忽略，那么最后进入人眼中光的 RGB 值则为(LR×MR, LG×MG, LB×MB)。

类似地，如果有两束光(R1, G1, B1)和(R2, G2, B2)进入人眼，OpenGL 将它们合成为

(R1+R2, G1+G2, B1+B2)。和大于 1（表明这种颜色超过可能显示的最大亮度）时，将该值截取为 1。

7.2　一个简单的例子：光照球体的渲染

下面列出的是向场景中增加光照需要的步骤：

1）为所有物体的顶点定义法线向量。这些法线决定了物体相对于光源的方位。

2）创建、选择光源，并为光源定位。

3）创建并选择光照模型，定义全局环境光的等级和观察点的有效位置（用于光照计算）。

4）为场景中的物体定义材料属性。

例 7-1 按部就班地完成上述各个步骤。在例子中，渲染一个被光源照亮的球体，如图 7-1 所示。

例 7-1　绘制一个被照亮的球体

```
#include "windows.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void)
{
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 100.0 };
  GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glDepthFunc(GL_LEQUAL);
  glEnable(GL_DEPTH_TEST);
}

void CALLBACK display(void)
{
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    auxSolidSphere(1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
            1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
        glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
            1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 7-1");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return 0;
}
```

与光照有关的调用在 myinit()函数中；下面将简要讨论这些调用，并在本章后面详细说明。

注意：例 7-1 采用的是 RGBA 颜色模式，而不是颜色索引模式。OpenGL 对两种模式所执行的光照计算是不同的。在颜色索引模式中实现光照的性能要受更多的限制，因此我们推荐读者进行光照编程时最好选用 RGBA 颜色模式。本章中全部例子都是用的 RGBA 颜色模

式。更详细的关于光照的颜色索引模式的信息，请参阅“光照的颜色索引模式”一节。

7.2.1 为所有的物体顶点定义法线向量

一个物体的法线决定了该物体相对光源的方向。并且，OpenGL 通过指定法线来确定各个顶点从每个光源接收到的光线数量。在例 7-1 中，球体的法线是在 auxSolidSphere()函数中定义的。关于如何定义法线的详细信息，请参见“法线向量”一节。

7.2.2 创建、定位和激活一个或多个光源

例 7-1 仅用到了一个白色光源，位置由函数 glLightfv()指定。在该例中，光源的颜色使用了缺省值（GL_LIGHT0），表示白色；若想获取其他颜色的光，可以使用 glLight*()来指定。在场景中至多可以包含八个不同颜色的光源，再多的话将被缺省地置为黑色（某些 OpenGL 实现也可能允许多于八个光源）。光源可以设置在任何期望位置——可以设置得很靠近场景，模拟一盏台灯的作用；也可以设置在无穷远处，模仿太阳。另外，还可以设定光源来产生一束窄而集中的光线或者是宽而发散的光线。

注意：每增加一个光源，都将大大增加场景渲染的计算量，从而降低程序的执行速度。

对创建具有期望特征的光源的详细介绍，请参见“创建光源”一节。

定义了期望的光源特征之后，便可以使用参数 GL_LIGHTING 调用函数 glEnable()，让 OpenGL 开始执行光照计算。关于上述过程的详细说明，请参阅“激活光源”一节。

7.2.3 选择光照模型

函数 glLightModel*()描述了光照模型的参数。在例 7-1 中，光照模型中惟一进行了明确定义的成分是全局环境光。光照模型中还可以定义观察者相对于场景的位置（较远还是较近），以及是否需要对场景中物体的前后面执行不同的光照计算。本例中这两个模型参数采用了缺省设置，使观察者位于无穷远处，且只进行单侧光照计算。一方面，近距离观察将大大增加 OpenGL 计算的复杂性，因为此时 OpenGL 必须计算视点与每个物体间的角度。而设定观察者在无穷远处，这个角度就可以忽略，并且不会引起失真。另一方面，在此例中，由于球面的背面不可见，所以只需要进行单面的光照计算。在“选择光照模型”一节中对 OpenGL 的光照模型参数进行了详细描述。

7.2.4 定义场景中物体的材质属性

一种物体的材质属性决定了它如何反射光线，也由此决定了该物体表现出来的材质属性。物体的材质表面与入射光的相互作用是非常复杂的，所以只有通过大量的编程实践积累丰富的经验，才能熟练地通过设定材质属性来使物体表现出期望的外观。另外，材质的环境颜色、散射颜色、镜面反射颜色和亮度也可以同时进行设定。在本例中，仅通过函数 glMaterialfv()设定了镜面反射颜色和亮度两个材质属性参数。“定义材料属性”一节中描述

了所有材质属性的参数，并给出了一个实例。

7.2.5 一些重要说明

在编写用户的光照程序的时候，一定要记住，只有部分光照参数可以使用缺省值，另一些则需要变动。另外，已经定义的光源和光照计算都需要被激活。最后还要记住，光照条件发生改变时，使用显示列表可以最大限度地提高效率。请参阅“显示列表设计准则”一节。

7.3 创建光源

光源有几个特征参数，如颜色、位置和方向。下面解释对这些参数的控制，及其对光源的影响。用于指定光源属性参数的函数是 glLight*()，它有三个参数，分别标识需要指定属性的光源、光源属性和期望的属性值。

---

void glLight{if}[v](GLenum light, GLenum pname, TYPEparam);

创建由 light 指定的光源，它可以是 GL_LIGHT0， GL_LIGHT1， ... ，或 GL_LIGHT7。光源属性由 pname 定义，它指定了一个命名参数（见表 7-1）。param 参数为 pname 指定了光源的属性值，在函数的向量形式下，它是一个指向数组的指针，非向量形式下是一个数值。

注意：只有在设定单值光源属性时才可以用到非向量形式。

表 7-1 glLight*()函数 pname 参数的缺省值

参数名	缺省值	意义
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	光源的环境光亮度
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	光源的散射光亮度
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	光源的镜面反射光亮度
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	光源的位置(x, y, z, w)
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	聚光方向(x, y, z)
GL_SPOT_EXPONENT	0.0	聚光指数
GL_SPOT_CUTOFF	180.0	聚光终止角度
GL_CONSTANT_ATTENUATION	1.0	恒定衰减因子
GL_LINEAR_ATTENUATION	0.0	线性衰减因子
GL_QUADRATIC_ATTENUATION	0.0	二次衰减因子

注意：表 7-1 中列出的 GL_DIFFUSE 和 GL_SPECULAR 的缺省值只适用于 GL_LIGHT0。对其他的光源，这两个参数缺省都取(0.0，0.0，0.0，1.0)。

下面是使用 glLight*()的例子。

```
GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

在例中，程序把参数值定义成数组，然后反复调用 glLightfv()来设定不同的参数。本例中 glLightfv()的前三个调用是多余的，因为它们给 GL_AMBIENT，GL_DIFFUSE 和 GL_SPECULAR 设定的还是缺省值。

注意： *在程序编制中一定要记得用 glEnable()函数将各个光源打开。参阅“激活光照”一节。*

下节将介绍 glLight*()函数的所有参数及其可能值。这些参数将与定义了场景光照模型的其他参数及物体材料属性发生相互作用。在“选择光照模型”和“定义材质属性”中分别对这两个主题进行了详细的讨论。“光照的数学计算”一节则从数学上解释了这些参数的相互作用。

7.3.1　颜色

OpenGL 允许为任何光源设置三个与颜色相关的参数 GL_AMBIENT，GL_DIFFUSE 和 GL_SPECULAR。GL_AMBIENT 参数是指加入场景中的光源中环境光的 RGBA 强度。从表 7-1 可以看出缺省情况下 GL_AMBIENT 的值是（0.0，0.0，0.0，1.0），即没有环境光。例 7-1 中用的就是该值。

如果该程序按照如下过程指定了蓝色环境光：

```
GLfloat light_ambient[] = { 0.0, 0.0, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
```

结果将如彩图 14a 中的左半部分所示的那样。

参数 GL_DIFFUSE 的作用最接近于直观想象中的“光的颜色”。它定义了加入场景的光源中散射光的 RGBA 颜色。缺省情况下，GL_LIGHT0 的 GL_DIFFUSE 值为（1.0，1.0，1.0，1.0），将产生一种如彩图 14a 所示的亮白光。对于其他光源（GL_LIGHT1, ... , GL_LIGHT7），GL_DIFFUSE 的缺省值是（0.0, 0.0, 0.0, 0.0）。

参数 GL_SPECULAR 影响着物体上镜面反射区域的颜色。通常，真实世界中的物体，如一只玻璃瓶等，其镜面反射区的颜色就是射来的光线颜色。因此为了产生接近真实世界的效果，将 GL_SPECULAR 与 GL_DIFFUSE 的值设为相同的值。缺省情况下，光源 GL_LIGHT0

的参数 GL_SPECULAR 值为(1.0, 1.0, 1.0, 1.0)，对于其他光源，其缺省值为(0.0, 0.0,0.0, 0.0)。

7.3.2 定位和衰减

前面已经提到，光源的位置可以设置为无穷远，也可以离场景很近。这种情况称为方向型光源，它的作用是产生射向物体的平行光线，现实生活中的太阳光就是这种类型的光源。第二种情况称为位置型光源，它在场景中的确切位置将决定它对场景的影响，尤其要决定光线的方向。台灯就是位置型光源最典型的例子。从彩图 14c 中我们可以看出这两种光源的区别。

例 7-1 中使用的光源是一种方向型光源。

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

还可以为参数 GL_POSITION 提供一个四值向量(x, y, z, w)。若最后一个值 w 为 0，那么相应的光源就是方向型光源，(x, y, z)值表示它的方向。此方向以法线向量的形式，根据模型视图矩阵进行转换。GL_POSITION 的缺省值为(0, 0, 1, 0)，定义了一个方向型光源，其方向指向负 z 轴。

注意：指定一个方向为(0, 0, 0)的方向型光源虽然不会产生错误，但这个光源对场景没有任何贡献，因而是无意义的。

如果 w 值不为 0，光源就是位置型的，(x, y, z)值表示光源在物体坐标系中的位置（参阅附录 7)。这个位置根据模型视图矩阵进行转换并存储在人眼坐标系中。如何控制光源位置转换的详细讨论，请参见“控制光源的位置与方向”一节。缺省情况下，位置型光源的光线向各个方向发射，但也可以通过将其定义成聚光灯来获得锥体形的光柱。后面我们专辟了一节“聚光灯”来解释怎样定义聚光灯光源。

注意：具有平滑阴影的多边形表面的颜色是根据对各顶点的颜色计算决定的。如果多边形很大，而位置型光源被设置在距多边形很近的中间部位，顶点则可能因距离光源较远而收到较少的光线，这样整个多边形看起来会比想象中的暗。为了避免这种情况，需要尽量将大多边形分解成小多边形块。

对真实世界的光线来说，光的强度会随着距离的增加而衰减。由于方向型光源位于无穷远处，随距离衰减是没有任何意义的，所以它的衰减被设置为失效。然而，对位置型光源就需要设置衰减率。OpenGL 通过将光源产生的贡献乘以一个衰减系数使光的强度衰减。

注意：环境光，散射光和镜面反射光的贡献均是衰减的。只有发射光和全局环境光不衰减。

7.3.3 聚光灯

前面曾经提到，通过限定发射光线的形状，可以将位置型光源设置成一个聚光灯。为了创建聚光灯，需要确定光锥的伸展形状。

注意：聚光灯是位置型光源，所以必须指定其位置。方向型的聚光灯虽然也可以有定义，但这样做将导致无法预料的结果。

参数 GL_SPOT_CUTOFF 可以设定圆锥体轴线与圆锥体母线的夹角。锥体顶角是这个数值的两倍，如图 7-2 所示。

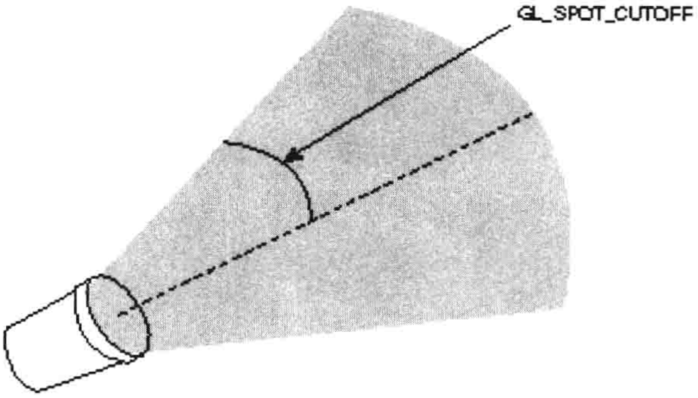


图 7-2　参数 GL_SPOT_CUTOFF

注意：在圆锥体母线之外是没有任何光线的。缺省情况下，由于参数 GL_SPOT_CUTOFF 取值为 180.0，此时意味着光线向各个方向发射出去（圆锥体顶点处的角度是 360 度，这已经不是圆锥体了），所以聚光特性并不能体现出来。

除了取特殊值 180.0 度，在体现聚光特性时，需要将 GL_SPOT_CUTOFF 的值限定在[0.0, 90.0]范围内。下面一行将 cutoff 参数设定为 45 度：

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0)
```

还需要指定聚光灯的方向，从而决定了光锥的轴线方向：

```
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

此方向在物体坐标系中进行指定。缺省的方向是（0.0, 0.0, -1.0），这样如果不明确设置 GL_SPOT_DIRECTION 的数值，光线将指向负 z 轴。

注意：聚光灯的方向是以法线向量的形式由模型视图矩阵转换而来的，转换的结果将存储在人眼坐标系中。详细的转换过程，请参阅“控制光源的位置和方向”一节。

除了聚光灯的角度和方向，还有两种方式控制圆锥体内光的强度分布。第一种方法是事先设定衰减系数，用它乘以光的强度。另外，可以通过设置参数 GL_SPOT_EXPONENT 来控制各光线向轴线聚集的程度，此参数缺省值为 0。光的强度在锥形体的轴线处为最大。从轴线到母线逐渐衰减，由此便引出聚光指数的概念。较高的聚光指数意味着光源有着较好的聚焦效果。关于光强方程的计算，请参阅“光照的数学计算”一节。

7.3.4　多光源

前面已经提到，我们所讨论的 OpenGL 场景中，至多可以使用八个光源。由于 OpenGL

要计算每个顶点从各个光源收到的光线，所以光源个数的增加对 OpenGL 应用性能有着不利影响。标识光源的常数可依次取为 GL_LIGHT0, GL_LIGHT1, GL_LIGHT2, GL_LIGHT3…依此类推。在前面的讨论中，已经设定了参数 GL_LIGHT0。若需要加入其他光源，就需要为新光源设定参数。增加的光源的缺省参数与 GL_LIGHT0 的缺省值是不同的，参见表 7-1 中对这些参数及缺省值的详细解释。下面的代码定义了一个白色的衰减聚光灯：

```
GLfloat light1_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
GLfloat light1_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light1_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light1_position[] = { -2.0, 2.0, 1.0, 1.0 };
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };

glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, light1_specular);
glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
glLightf(GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.5);
glLightf(GL_LIGHT1, GL_LINEAR_ATTENUATION, 0.5);
glLightf(GL_LIGHT1, GL_QUADRATIC_ATTENUATION, 0.2);
glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 45.0);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 2.0);
glEnable(GL_LIGHT1);
```

将上述代码加入例 7-1 中，读者会得到这样的运行结果：球体将被两个光源照亮，一个为方向型光源，一个为聚光灯。

试一试

修改例 7-1：

- 将方向型白光光源改成一种有颜色的位置型光源。
- 增加一个有颜色的聚光灯。提示：利用前一节中的代码。
- 观察这两种改变对性能的影响。

7.3.5 控制光源的位置与方向

OpenGL 对光源位置、方向的处理与对几何图元一样。换句话说，光源同图元遵循相同的矩阵变换。调用函数 **glLight*()**来指定光源的位置与方向时，位置和方向要由当前模型视图矩阵进行转换，并存储在人眼坐标系中。这就意味着，通过改变模型视图矩阵堆栈的内容可以对光源的位置和方向进行操作（投影矩阵对光源的位置与方向没有影响）。这一节将解

释怎样通过改变程序中对光源位置的设置，来实现与模型变换和视图变换相关的三种不同效果。

- 位置固定的光源。
- 围绕静止物体移动的光源。
- 沿视点方向移动的光源。

像例 7-1 这样较简单的例子中，光源的位置是固定的。这时只需要在视图和/或模型变换后设置光源的位置。下面是 myInit() 和 myReshape()函数中的相关代码。

```
glViewport(0, 0, w, h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
if (w <= h)
   glOrtho (-1.5, 1.5, -1.5*h/w, 1.5*h/w, -10.0, 10.0);
else
   glOrtho (-1.5*w/h, 1.5*w/h, -1.5, 1.5, -10.0, 10.0);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity();

/* 后面的myInit()中： */
GLfloat light_position[] = { 1.0, 1.0, 1.0, 1.0 };
glLightfv(GL_LIGHT0, GL_POSITION, position)
```

在上述代码中，首先建立了视图矩阵和投影矩阵。然后将单位阵加载到模型视图矩阵中，再设置光源位置。单位阵的使用，使最初指定的光源位置（1.0, 1.0, 1.0）在乘以模型视图矩阵后不发生改变。这样一来，光源位置和模型视图矩阵都未改变，光源仍保持在（1.0，1.0，1.0）处。

如果要将光源进行平移或旋转，从而产生相对于静止物体的运动，可以在模型变换之后设定光源位置，这样便明确地改变了光源位置。上述程序中 init()函数的前面部分不变，但在事件循环中需要进行期望的模型变换（在模型视图堆栈上），重新设置光源位置。下面是这一部分的代码：

```
void CALLBACK display(GLint spin)
 {
  GLfloat light_position[] = { 0.0, 0.0, 1.5, 1.0 };
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  glTranslatef(0.0, 0.0, -5.0);
  glPushMatrix();
```

```
    glRotated((GLdouble) spin, 1.0, 0.0, 0.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glPopMatrix();
    auxSolidTorus();
    glPopMatrix();
    glFlush();
}
```

display()函数对场景进行了重画，光源将沿着一个静止的圆环旋转 spin 度。注意，程序中调用了两对 glPushMatrix()和 glPopMatrix()函数，用来隔开视图变换和模型变换，这些变换都在模型视图堆栈上进行。本例中由于视点保持不变，因此当前矩阵被压入堆栈中，然后由 glTranslatef()函数执行期望的视图变换。在模型变换 glRotated()执行之前，矩阵再一次被压入堆栈。然后，在旋转后的新坐标系中重新设置光源位置，使光源相对于原来的位置也产生一个旋转（光源位置存储在经过了模型视图矩阵变换后的人眼坐标中）。旋转矩阵弹出堆栈后，圆环就画出来了。

为了创建沿视点移动时的光源效果，需要在视图变换之前设置光源位置。视图变换对光源和视点便会产生同样的影响。下面我们在上例的 myInit()函数中，对调用作一些小小的改变。

```
GLfloat light_position() = { 0.0, 0.0, 1.0, 1.0 };
glViewport(0, 0, w-1, h-1);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 100.0);
glMatrixMode(GL_MODELVIEW);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

然后，在事件循环中调用重画场景的 display()函数，其代码如下：

```
void CALLBACK display(GLint spin)
{
    glClear(GL_COLOR_BUFFER_MASK | GL_DEPTH_BUFFER_MASK);
    glPushMatrix();
        glTranslatef (0.0, 0.0, -5.0);
        glRotatef ((GLfloat) spin, 1.0, 0.0, 0.0);
        auxSolidTorus();
    glPopMatrix();
    glFlush();
}
```

当圆环重画时，光源位置和视角都旋转了 spin 度。尽管并未重新指定光源位置，但由于人眼坐标系的改变，光源也跟着发生了移动。

试一试

修改例 7-2：

- 使光源平移经过物体，而不是绕其旋转。提示：用 glTranslated()代替 display()中的 glRotated()函数，并选择一个适当步长值代替转角 spin。
- 选择衰减系数，使光源离开物体时光的强度逐渐衰减。提示：增加 glLight*()调用来设置期望的衰减系数。

例 7-2　通过模型变换移动光源：movelight.c

```
#include "windows.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

static int spin = 0;

void CALLBACK movelight (AUX_EVENTREC *event)
{
  spin = (spin + 10) % 360;
}

void myinit (void)
{
 glEnable(GL_LIGHTING);
 glEnable(GL_LIGHT0);
 glDepthFunc(GL_LEQUAL);
 glEnable(GL_DEPTH_TEST);
}

void CALLBACK display(void)
{
  GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix ();
```

```
  glTranslatef (0.0, 0.0, -5.0);
  glPushMatrix ();
  glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);
  glRotated (0.0, 1.0, 0.0, 0.0);
  glLightfv (GL_LIGHT0, GL_POSITION, position);
  glTranslated (0.0, 0.0, 1.5);
  glDisable (GL_LIGHTING);
  glColor3f (0.0, 1.0, 1.0);
  auxWireCube (0.1);
  glEnable (GL_LIGHTING);
  glPopMatrix ();
  auxSolidTorus (0.275, 0.85);
  glPopMatrix ();
  glFlush ();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
  glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 7-2");
    myinit();
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, movelight);
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
     return 0;
```

```
}
```

运行结果如图 7-3 所示。

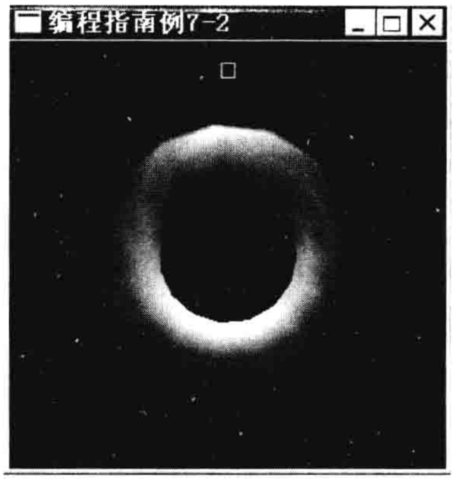


图 7-3　例 7-2 的运行结果

7.4　选择光照模型

OpenGL 的光照模型概念包括三个部分：

- 全局环境光强度。
- 观察点的位置是靠近场景还是位于无穷远处。
- 对物体的正面和背面是否采取相同的光照计算。

本节解释了如何指定一个光照模型，并讨论了如何激活光照（即通知 OpenGL 开始进行光照计算）。

7.4.1　全局环境光

如前所述，每一个光源都将给场景带来一定的环境光。另外，还可能存在与任何光源都无关的环境光。为了指定全局环境光的 RGBA 强度，应该这样设置参数 GL_LIGHT_MODEL_AMBIEN：

```
GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

在本例中，lmodel_ambient 取 GL_LIGHT_MODEL_AMBIENT 的缺省值。这种情况下，将产生微量的白色环境光，所以即使场景中没有其他任何光源，也可以看到场景中的物体。彩图 15a 显示了不同量全局环境光的效果。

7.4.2　局部或无穷远的视点

视点的位置影响着镜面反射区的计算。更确切地说，一个特定顶点的镜面反射亮度取决

于该顶点的法线、顶点相对光源的方向、以及顶点相对视点的方向。记住，视点实际上并未因为调用了光照函数而移动（只有改变投影变换才可以真正移动视点）。光照计算是假设视点的位置移动了。

在无穷远视点的情况下，场景中任何顶点到视点的方向都是同一个常数，这样计算和渲染十分方便快捷。局部视点的效果更真实，但由于需要单独计算每个顶点的方向，性能将大打折扣。视点的缺省值是设置在无穷远处。下面的代码例示了如何将视点设为局部视点。

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

该函数将视点置于人眼坐标系中的（0, 0, 0）点。如果要将视点重新置于无穷远，以GL_FALSE 作为参量调用此函数即可。

7.4.3 双面光照

无论是多边形正面还是背面，都无一例外地需要执行光照计算。在例 7-1 中，由于物体是一封闭的球体，多边形只有正面是可见的。这种情况下，没有必要去考虑球体内部是什么样子，所以单面光照已经足够了。但是，如果球体被切开使内表面可见，就要定义光照条件使内表面被照亮，同时给出对内表面的材质特征描述。按下面的代码可以实现双面光照：

```
glLightModeli(LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

OpenGL 将背面多边形的表面法线取反，这样，可见的背面多边形的表面法线同正面多边形的一样，是指向观察者的，所有多边形的正面和背面便都可以方便地实现光照效果。

在上述调用中取参数 GL_FALSE，即可将双面光照关闭。对两个表面材料属性设置的详细介绍，请参阅“定义材料属性”一节。通过 glFrontFace()函数可以控制 OpenGL 中哪个表面是正面。

7.4.4 激活光照

在 OpenGL 中，必须明确指定光照状态是激活还是失效。未激活光照时，OpenGL 仅将当前颜色简单地映射到当前矩阵中，而法线、光源、光照模型和材质属性等计算都不执行。下面的函数用于激活光照：

```
glEnable(GL_LIGHTING);
```

以 GL_LIGHTING 为参数调用 glDisable()函数便可以使光照失效。

指定了光源参数后，需要明确激活每个已有定义的光源。例 7-1 仅用到了一个光源GL_LIGHT0:

```
glEnable(GL_LIGHT0);
```

7.5 材料属性的定义

前面已经介绍了如何创建某种特征的光源，以及如何定义期望的光照模型。本节将描述

如何定义场景中物体的材料属性：环境颜色、散射颜色、镜面反射颜色、亮度和发射光的颜色。光照和材料属性计算所用的方程将在“光照的数学计算”一节中介绍。大多数材料属性在概念上与创建光源时用到的特征类似。除使用函数 glMaterial*()之外，设置这些参数的机制也类似。

---

void glMaterial{if}[v](GLenum face, GLenum pname, TYPEparam);

指定用于执行光照计算的当前材料属性。参数 face 分别可以取 GL_FRONT、GL_BACK 或 GL_FRONT_AND_BACK，表示材料属性应用于物体的哪个表面。所设置的材料属性由 pname 标识，属性值由 param 给出，param 可以是指向数组的指针（若属性值为一组数据），也可能是数值形式（若属性值是一个值）。只有在设置 GL_SHININESS 时才使用数值形式。

参数 pname 可能取到的值列于表 7-2 中。注意，使用 GL_AMBIENT_AND_DIFFUSE 时可以同时将材质的环境颜色和散射颜色设置为同一个 RGBA 值。

表 7-2　glMaterial*()中参数 pname 的缺省值

参数名	缺省值	意义
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	材质的环境颜色
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	材质的散射颜色
GL_AMBIENT_AND_DIFFUSE		材质的背景颜色和散射颜色
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	材质的镜面反射颜色
GL_SHININESS	0.0	镜面反射参数
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	材质的放射光颜色
GL_COLOR_INDEXES	(0,1,1)	背景颜色，散射颜色和镜面反射颜色的颜色索引

在“选择光照模型”一节中曾提到，物体的正面和背面可以分别采取不同的光照计算。若背面是可见的，则需要用函数 glMaterial*()的 face 参数对正面和背面分别取不同的材质属性。在彩图 15b 中，物体的内外表面具有不同的材质属性。

通过改变材质属性可以实现不同的效果，这一点在彩图 16 中可以清楚地看到。图中显示了同一物体在几组不同材质属性下的图像，图中使用了相同的光源和光照模型。下一节中将讨论每个球体所采用的材质属性。

注意：函数 glMaterial*()设置的大部分材质属性是以（R, G, B, A）颜色表示的。不论其他参数的 alpha 值是多少，顶点的 alpha 值就是材质的散射颜色的 alpha 值，即由函数 glMaterial*()设置 GL_DIFFUSE 的 alpha 值。材质属性的 RGBA 值不能用于颜色索引模式。关于颜色索引模式下相关参数更详细的介绍，请参见“颜色索引模式下的光照”一节。

7.5.1 散射和环境光反射

由函数 glMaterial*()设置的参数 GL_DIFFUSE 和 GL_AMBIENT 决定了物体散射光和环境光的颜色。

物体看起来是何种颜色，很大程度地受到散射光的影响。它取决于入射光中的散射光颜色和入射光相对于物体表面法线的角度。当入射光垂直于表面时散射光最强。视点的位置对散射光没有影响。

环境光反射也影响着整个物体的颜色，光直射物体时散射光最强，非直射时环境光的效果最明显。一个物体的全部环境光反射包括它对全局环境光的反射和对每个光源产生环境光的反射。同散射光的反射一样，环境光反射不受视点位置的影响。

对于真实世界中的物体，其散射光与环境光通常是同样颜色的。出于这种考虑，OpenGL 提供了一种简便的方法，利用 glMaterial*()函数将两个参数设为相同值：

```
GLfloat mat_amb_diff[] = { 0.1, 0.5, 0.8, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, mat_amb_diff);
```

在本例中，正面和背面的多边形都是采用 RGBA 值为（0.1, 0.5, 0.8, 1.0）的深蓝色来表示环境颜色和散射颜色。

彩图 16 中，第一行球没有环境颜色，即环境颜色为（0.0, 0.0, 0.0, 0.0），而第二行球有较强的环境颜色（0.7, 0.7, 0.7, 1.0）。

7.5.2 镜面反射

物体的镜面反射会在物体表面产生一个高亮区。与环境光和散射光的反射不同，观察者所看到的镜面反射依赖于视点的位置——沿着反射光的方向亮度最高。这一点是符合真实生活的。比如在户外的阳光下观察一只金属球，当视点移动时，阳光照射产生的高亮区也随之移动，并且当视点移到某一位置时，高亮区会消失。

OpenGL 中可以通过相应参数对镜面反射高亮区的各种属性进行设置：参数 GL_SPECULAR 可以设置 RGBA 颜色， GL_SHININESS 可以设置高亮区域的大小和亮度。GL_SHININESS 的取值范围为[0.0, 128.0]，数值越大，高亮区越小、亮度越高（光线更集中）。镜面反射高亮区的计算请参阅“光照的数学计算”一节。

彩图 16 中，第一列球体没有设置镜面反射属性。第二列球体的 GL_SPECULAR、GL_SHININESS 值分别计算如下：

```
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat low_shininess[] = { 5.0 };
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
```

到第三列球体，GL_SHININESS 值已增大到了 100.0。

7.5.3 发射光颜色

通过为 GL_EMISSION 设置 RGBA 颜色，可以使一个物体发出此种颜色的光。OpenGL 的这一功能常用来模拟场景中的灯等各种光源。

在彩图 16 中，第四列球体的 GL_EMISSION 值便被设置为绿色：

```
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```

注意：本图例中，看起来球体在微微发光，但是真实世界中的物体实体是起不到光源作用的。这里模拟的是创建一个光源并把它放置在与球体的位置上所产生的效果。

7.5.4 改变材质属性

例 7-1 中将场景中惟一的物体——球体的所有顶点设置为同一材质属性。有时，需要将同一物体的不同顶点设置为不同的材质属性。甚至在更一般的情况下，场景中有多个物体，每个物体都有不同的材质属性，同时每个物体的每个顶点都有自己的材质属性。生成彩图 16 所用的代码需要绘制 12 个不同物体（都是球体），每一个物体都有不同的材质属性。例 7-3 给出了 display()函数中的部分代码。

例 7-3　使用不同的材质属性：material.c

```
GLfloat no_mat[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat mat_ambient[] = { 0.7, 0.7, 0.7, 1.0 };
GLfloat mat_ambient_color[] = { 0.8, 0.8, 0.2, 1.0 };
GLfloat mat_diffuse[] = { 0.1, 0.5, 0.8, 1.0 };
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat no_shininess[] = { 0.0 };
GLfloat low_shininess[] = { 5.0 };
GLfloat high_shininess[] = { 100.0 };
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
/* 在第一行第一列绘制球体*/
/* 仅有散射，无环境光和镜面反射*/
glPushMatrix();
glTranslatef (-3.75, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
```

```
glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
auxSolidSphere(); glPopMatrix();
/* 在第一行第二列绘制球体*/
/* 有散射和镜面反射，低亮度，无环境光*/

glPushMatrix();
glTranslatef (-1.25, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
auxSolidSphere(); glPopMatrix();
/* 在第一行第三列绘制球体*/
/* 有散射和镜面反射，高亮度，无环境光*/

glPushMatrix();
glTranslatef (1.25, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
auxSolidSphere(); glPopMatrix();
/* 在第一行第四列绘制球体*/
/* 有散射反射，发光，无环境光和镜面反射*/

glPushMatrix();
   glTranslatef (3.75, 3.0, 0.0);
   glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
   glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
   glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```

```
    auxSolidSphere();
glPopMatrix();
```

可以看出，程序中反复调用 glMaterialfv()函数来设置各个球体的不同材质属性。

注意：只有需要更改材质属性时才调用此函数。第二和第三个球体与第一个球体使用了相同的环境和散射属性，所以这些属性不用重新设置。由于 glMaterial*()函数的调用很费时间，所以应尽量减少改变材质特征的次数。

另一个用于改变材质属性的函数是 glColorMaterial()，这个函数的调用对性能的影响并不是很大。

---

void glColorMaterial(GLenum face, GLenum mode)

此函数通过由 face 指定的材质表面及 mode 指定的材料属性来跟踪当前颜色值。当前颜色（使用 glColor*()函数）的变化将立即引起指定材质属性的变化。参数 face 可以取 GL_FRONT，GL_BACK，或 GL_FRONT_AND_BACK（缺省值）。参数 mode 可以取 GL_AMBIENT，GL_DIFFUSE，GL_AMBIENT_AND_DIFFUSE（缺省值），GL_SPECULAR 或 GL_EMISSION。

注意：glColorMaterial()函数指定了两个独立的值。第一个值说明了需要对哪个表面进行更新，第二个值指定对该表面上的哪种材质属性进行更新。OpenGL 并不为每个表面单独保存 mode 变量。

在 glColorMaterial()调用之后，还必须以 GL_COLOR_MATERIAL 作为参数调用 glEnable()函数。然后，就可以根据绘图需要利用 glColor*()函数来改变当前颜色（或调用 glMaterial*()函数来改变其他材质属性）。

```
glColorMaterial(GL_FRONT, GL_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);
glColor3f(0.2, 0.5, 0.8);
/* 此处绘制一些物体 */
glColor3f(0.9, 0.0, 0.2);
/* 此处绘制另一些物体 */
glDisable(GL_COLOR_MATERIAL);
```

当需要改变场景中大部分顶点的各个材质参数时，应使用 glColorMaterial()函数。如果像彩图 16 那样，需要改变部分材质参数，可以使用 glMaterial*()函数。

注意：当不再需要使用 glColorMaterial()函数时，一定要使其失效，否则可能导致不希望的材质属性变化，还会使性能降低。glColorMaterial()对应用程序性能的影响与具体 OpenGL 实现有关。在某些实现中可能对顶点函数进行了优化，这样就可以在当前颜色的基础上快速更新材质属性。

例 7-4 给出了使用 glColorMaterial()函数来改变材质参数的交互式程序。按下鼠标键中的任何一个都会改变散射光的颜色。

例 7-4 使用 glColorMaterial()函数: colormat.c

```
#include "windows.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

GLfloat diffuseMaterial[4] = { 0.5, 0.5, 0.5, 1.0 };

void myinit(void)
{
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuseMaterial);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 25.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
    glColorMaterial(GL_FRONT, GL_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);
}

void CALLBACK changeRedDiffuse (AUX_EVENTREC *event)
{
    diffuseMaterial[0] += 0.1;
    if (diffuseMaterial[0] > 1.0)
         diffuseMaterial[0] = 0.0;
    glColor4fv(diffuseMaterial);
}
```

```
void CALLBACK changeGreenDiffuse (AUX_EVENTREC *event)
{
    diffuseMaterial[1] += 0.1;
    if (diffuseMaterial[1] > 1.0)
        diffuseMaterial[1] = 0.0;
    glColor4fv(diffuseMaterial);
}

void CALLBACK changeBlueDiffuse (AUX_EVENTREC *event)
{
    diffuseMaterial[2] += 0.1;
    if (diffuseMaterial[2] > 1.0)
        diffuseMaterial[2] = 0.0;
    glColor4fv(diffuseMaterial);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    auxSolidSphere(1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
      glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
    5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
      glOrtho (-1.5*(GLfloat)w/(GLfloat)h,  1.5*(GLfloat)w/(GLfloat)h,
               -1.5, 1.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
```

```
    glLoadIdentity();
}

int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
   auxInitPosition (0, 0, 300, 300);
   auxInitWindow ("编程指南例 7-4");
   myinit();
   auxMouseFunc(AUX_LEFTBUTTON, AUX_MOUSEDOWN,
      changeRedDiffuse);
   auxMouseFunc(AUX_MIDDLEBUTTON, AUX_MOUSEDOWN,
      changeGreenDiffuse);
   auxMouseFunc(AUX_RIGHTBUTTON, AUX_MOUSEDOWN,
      changeBlueDiffuse);
   auxReshapeFunc(myReshape);
   auxMainLoop(display);
    return 0;
}
```

试一试

修改例子 7-3：

- 改变场景中的全局环境光。提示：改变参数 GL_LIGHT_MODEL_AMBIENT 的值。
- 改变散射颜色、背景颜色、镜面反射颜色、亮度指数及发射光颜色。提示：使用 glMaterial*()函数，但要避免调用次数过多。
- 使用双面材质并增加一个任意的裁剪面（参见“附加裁剪平面”一节），这样可以看到球体的里、外表面。提示：以 GL_LIGHT_MODEL_TWO_SIDE 打开双面光照，设置期望的材质属性，并增加裁剪面。
- 删除所有的 glMaterialfv()调用，以效率更高的 glColorMaterial()函数实现相同的光照效果。

7.6 光照的数学计算

本节给出了 RGBA 模式下，OpenGL 对颜色执行光照计算所用的方程。颜色索引模式下的相应计算参见“颜色索引模式下光照的数学计算”一节。本节的内容完全是数学理论上的，真正要掌握光照效果，并得到参数值对顶点颜色的影响中较深层次的理解，需要进行大

量的、长时间的编程实践和试验。

注意：若光源未被激活，则顶点的颜色就是当前颜色；如果激活了光源，这里所描述的光照计算将在人眼坐标系中执行。

在下面的方程中，对 R, G 与 B 各分量的计算是分别进行的。例如，下面列出的三项，每一项中的 R、G、B 值要分别相加，形成最后的 RGB 颜色（$R_1+R_2+R_3$，$G_1+G_2+G_3$，$B_1+B_2+B_3$）。如果三项相乘，计算结果则为（$R_1+R_2+R_3$，$G_1+G_2+G_3$，$B_1+B_2+B_3$）。

注意：最后的 A 值，即 alpha 值等于该顶点材质的散射 alpha 值。

照亮一个顶点产生的颜色按下式计算：

顶点颜色 = 该顶点发射光的颜色 +
　　由材质环境光属性放大的全局环境光颜色 +
　　从各个光源发出，经过衰减的环境光、散射光和镜面反射光颜色

光照计算完成以后，颜色值（*RGBA* 模式下）将被截取在区间[0,1]之内。

注意：OpenGL 的光照计算并没有考虑物体之间互相遮挡的可能性，因此不会自动地生成阴影。创建阴影的技术，请参见“阴影”一节。另外，还要记住一点，在 OpenGL 中，被照亮的物体并不向其他物体发射光线。

7.6.1　材质的发射光

材质的发射光最简单。它就是分配给 GL_EMISSION 参数的 RGB 值。

7.6.2　放大的全局环境光

第二项由全局环境光（由参数 GL_LIGHT_MODEL_AMBIENT 进行定义）与材质的环境光属性（由函数 glMaterial*()分配的 GL_AMBIENT 值）相乘得到：

$$\text{ambient}_{\text{light model}} \times \text{ambient}_{\text{material}}$$

在公式中，两个参数的相应 R、G、B 值两两相乘，得出这一项的最终的 RGB 值（R_1+R_2，G_1+G_2，B_1+B_2）。

7.6.3　光源的贡献

每个光源对顶点颜色均有贡献，这些贡献是叠加在一起的。计算每个光源的贡献可以按照下面的方程：

整个光源的效果 = 衰减系数 * 聚光灯效果 * (环境光项 + 散射光项 + 镜面反射光项)

7.6.4　聚光灯效果

聚光灯效果可能有三种取值，依赖于光源是否为聚光灯、顶点位于聚光灯光照范围之内还是之外。

- 如果光源不是聚光灯（GL_SPOT_CUTOFF 为 180.0），取值为 1。
- 如果光源是聚光灯，但顶点位于光照范围之外，那么取值为 0。
- 假设光源为聚光灯并且顶点位于光照范围之内，那么取值为：

$$(\max\ \{v\times d,\ 0\})^{GL_SPOT_EXPONENT}$$

其中：

$v = (v_x, v_y, v_z)$为从聚光灯（GL_POSITION）指向顶点的单位向量。

$d = (d_x, d_y, d_z)$为聚光灯的方向（GL_SPOT_DIRECTION）。

两个向量 v 和 d 的点积与它们夹角的余弦成正比，因此物体在轴线上的亮度最大，离开轴线将使亮度与夹角的余弦成比例下降。

OpenGL 用（max{v×d, 0}）来确定某个顶点是否位于聚光灯照明区间内，其中 v 和 d 的意义上面已经介绍。如果该值小于聚光灯终止角度（GL_SPOT_CUTOFF）的余弦值，则表明顶点位于照明区之外；否则，顶点位于照明区之内。

7.6.5 环境光项

环境光项就是光线中的环境光颜色和材质的环境光属性简单相乘的结果。

$$ambient_{light} \times ambient_{material}$$

7.6.6 散射光项

散射光项需要考虑光线是否照在该顶点上、入射光的散射光颜色、以及材质的散射光属性。

$$(\max\{l\times n, 0\}) \times diffuse_{light} \times diffuse_{material}$$

其中：

$l = (l_x, l_y, l_z)$ 是从顶点指向光源（GL_POSITION）的单位向量。

$n = (n_x, n_y, n_z)$ 是顶点的单位法线向量。

7.6.7 镜面反射光项

镜面反射光项也取决于光线是否照在该顶点上。如果l×n 小于或等于 0，则此顶点上没有镜面反射。l×n 小于 0 表明光线从表面的反方向射来。若存在镜面反射，它由以下因素决定：

- 顶点的单位法线向量(n_x, n_y, n_z)。
- 两个单位向量的和：①从顶点指向光源位置，②从顶点指向视点。这里假设 GL_LIGHT_MODEL_LOCAL_VIEWER 的值为真；否则，向量（0, 0, 1）将作为求和时的第二个向量。和向量需要进行归一化（每个分量除以向量的模），得到

  $s = (s_x, s_y, s_z)$。

- 镜面反射指数（GL_SHININESS）。
- 光的镜面反射颜色（$GL_SPECULAR_{light}$）。
- 材质的镜面反射属性（$GL_SPECULAR_{material}$）。

利用以上定义，OpenGL 按下式计算镜面反射光项：

$$(\max\{s\times n, 0\})^{shininess} \times specular_{light} \times specular_{material}$$

如果 $1\times n = 0$，则此项为 0。

7.6.8 求和

根据前面对各项的定义，下面给出了 RGBA 模式下的整个光照计算公式：

$$\begin{aligned} \text{vertex color} = {} & emission_{material} + \\ & ambient_{light\ model} * ambient_{material} + \\ & \sum_{i=0}^{n-1}\left(\frac{1}{K_c + K_l d + K_q d^2}\right)_i (spotlight_effect)_i \\ & ambient_{light} * ambient_{material} + \\ & (\max\{l\times n, 0\}) * diffuse_{light} * diffuse_{material} + \\ & (\max\{s\times n, 0\})^{shininess} \times specular_{light} \times specular_{material} \end{aligned}$$

7.6.9 颜色索引模式下的光照

在颜色索引模式下，包含 RGBA 值的参数不起任何作用，也没有特定意义。由于在颜色索引模式下很难实现某种效果，所以应尽可能使用 RGBA 模式。实际上，RGBA 模式的光源、光照模型或材质属性参数中，只有光源参数 GL_DIFFUSE 和 GL_SPECULAR 与材质参数 GL_SHININESS 可以用于颜色索引模式。下式用来计算颜色索引模式下的散射光和镜面反射光（d_{ci} 与 s_{ci}）亮度：

$d_{ci} = 0.30\ R(d_l) + 0.59\ G(d_l) + 0.11\ B(d_l)$

$s_{ci} = 0.3\ R(s_l) + 0.59\ G(s_l) + 0.11\ B(s_l)$

其中，R(x)，G(x)和 B(x)分别表示颜色 x 的红、绿、蓝成分。加权值 0.30，0.59 和 0.11 反映了人眼对红、绿、蓝色的“感觉”的权值。事实上，人眼对绿色最敏感、对蓝色最不敏感。

为了在颜色索引模式下设置材质的颜色，需要用特定的参数 GL_COLOR_INDEXES 来调用函数 glMaterial*()。

```
GLfloat mat_colormap[] = { 17.0, 47.0, 79.0 };
glMaterialfv(GL_FRONT, GL_COLOR_INDEXES, mat_colormap);
```

为 GL_COLOR_INDEXES 提供的三个数值分别用于指定材质的环境颜色（本例中为 17.0）、散射颜色（47.0）和镜面反射颜色（79.0）的颜色索引。缺省情况下，环境颜色为 0.0，

散射颜色和镜面反射颜色的索引值都是 1.0。

注意：glColorMaterial()函数对颜色索引模式下的光照不起作用。

在绘制场景时，OpenGL 要用到与索引值相关的颜色来处理场景中物体颜色的深浅。因此，必须在指定的索引值之间建立颜色梯度。在本例中，颜色梯度在 16 与 47 之间，及 47 与 79 之间。通常所建立颜色梯度是平滑的，但有时可以用其他公式产生不同的效果。下面是一个平滑颜色梯度的例子，该颜色梯度以黑色的环境光开始，经过了洋红色的散射光，最后是白色的镜面反射光。

```
for (i = 0; i < 32; i++) {
   auxSetOneColor (16 + i, 1.0 * (i/32.0), 0.0, 1.0 * (i/32.0));
   auxSetOneColor (48 + i, 1.0, 1.0 * (i/32.0), 1.0);
}
```

辅助库函数 auxSetOneColor()有四个参数。它将第一个参数指定的颜色索引值与后三个参数指定的 RGB 三元组联系起来。当 i = 0 时，给颜色索引 16 分配的 RGB 颜色值为(0.0, 0.0, 0.0)，即黑色。

所建立的平滑颜色梯度的最后颜色，是索引值 47（对应 i = 31）对应的散射光颜色，它的 RGB 颜色值为纯洋红色，即（1.0, 0.0, 1.0）。在第二个循环中，建立了从纯洋红色的散射光颜色到白色（1.0, 1.0, 1.0）镜面反射颜色（索引值为 79）的颜色梯度。

7.6.10 颜色索引模式下光照的数学计算

由于颜色索引模式下允许使用的参数与 RGBA 模式下不尽相同，因此对其计算也完全不同。在颜色索引模式下，任何材料都没有发射光和环境光，所以在 RGBA 方程中惟一感兴趣的项就是光源的散射，镜面反射的贡献和亮度。当然，这些项必须按下面的方式进行修改。

在 RGBA 方程中，是以散射及镜面反射开始的。在散射项中，用前一节为颜色索引模式定义的 d_{ci} 来代替 $diffuse_{light} \times diffuse_{material}$。类似地，对镜面反射项，要用前一节中定义的 s_{ci} 来代替 $specular_{light} \times specular_{material}$。衰减系数、聚光灯效果、以及其他各项分量按前面介绍的方法计算。调用时要用到这些经过修改的散射光项 d 和镜面反射光项 s。

现在，令 $s' = \min\{ s, 1 \}$，然后计算下式：

$$c = a_m + d(1-s')(d_m - a_m) + s'(s_m - a_m)$$

其中，a_m，d_m，和 s_m 是由 GL_COLOR_INDEXES 指定的材质环境颜色、散射颜色和镜面反射颜色的颜色索引。最终的颜色索引值为：

$$c' = \min\{ c, s_m \}$$

在光照计算完成以后，颜色索引值被转换为定点浮点数。然后用 2^n-1 将整数部分屏蔽掉（按位与“AND”运算），其中 n 是颜色索引缓存中的颜色的位数。

第 8 章　混合、反走样、雾

本章主题

在前面的各章节中已经论述了创建计算机图形场景所需的基本工具。通过前面的阅读，读者已经能够顺利完成下列工作：

- 绘制几何形体。
- 对几何体进行变换，从理想的角度观察它们。
- 利用显示列表优化程序性能。
- 确定如何对场景中的几何形体进行着色，以及如何产生明暗效果。
- 增加光照，说明光照如何影响场景中的几何形体。

应该说，到这一章，我们的学习进入了一个新的阶段。后面的内容，无论是程序编制还是原理学习，都会激发出读者更大的兴趣，并更高度地模拟真实世界。本章集中讨论的 3 种技术，便会使程序员创建的场景更加细腻，更加生动，而且这些技术都简单易用。它们的使用甚至比解释起来更容易。在下面的几节中，我们将分别讨论这些技术：

- 混合，讲述应如何指定混合函数，该函数将原始颜色与目标颜色结合起来。最终使场景中的某些部分产生半透明效果。
- 反走样，解释了这种相当微妙的技术，它通过改变颜色来消除点、直线，及多边形边缘的棱角和锯齿，使其看起来更平滑，更接近真实。
- 雾，描述了如何根据物体与视点的距离来计算物体的颜色值，以便创建一种深度错觉。雾的作用是使距视点较远的物体看起来好像融合到了背景之中。这个技术是对真实生活的一种逼近。

8.1　混合

在前面，已经对 alpha 值（RGBA 中的 A）有所接触，但值总是取 1.0，而且并未进行详细的讨论。利用函数 glClearColor()指定清除颜色，或指定诸如材质、光源强度等光照参数时，alpha 值是通过函数 glColor*()指定的。在第 6 章中已经学习过，显示器上的像素发出红、绿、蓝三种光，这些光的颜色是由红、绿、蓝颜色值来控制的。那么，alpha 值是如何影响屏幕窗口所画的物体的呢？

激活了混合操作后，alpha 值用来将已处理过的片原颜色值与存储在帧缓存中的像素颜色值结合起来。而混合操作的进行是在场景已经被光栅化并变换成片原之后，并且在最终像

素画入帧缓存之前。alpha 值还可用来进行 alpha 测试，基于测试的结果 alpha 值来确定应该接受还是舍弃某一片原。这一过程的详细说明请参见第 11 章。

如果不进行混合操作，每个新的片原都将被 OpenGL 当成是不透明的，覆盖掉帧缓存中已存在的所有颜色值。利用混合操作，可以控制将多少成分的既存颜色值与新片原颜色值进行合成。因此，通过 alpha 混合，显示部分先前存储的颜色，便可以创建半透明的片原。颜色混合技术是诸如透明度、数字合成、着色等技术的核心。

注意：*在颜色索引模式不需要指定 alpha 值。因此，颜色索引模式下不能进行混合操作。*

以一种自然的方式，可以这样来考虑混合操作，片原的 RGB 分量相当于对其颜色的表示，则 alpha 分量表示了颜色的不透明程度。因此，透明或半透明表面比不透明表面的不透明度低。例如，如在透过绿色玻璃观察一个物体时，所观察到的颜色会有几分玻璃的绿色，又有几分物体的颜色。绿色所占的百分比是随玻璃的透射性而变化的：如果射到玻璃上的光透过了 80%（即有 20%不透明），那么所看到的颜色就是 20%的玻璃颜色与 80%物体颜色的合成。

有时在场景中还会存在多个半透明表面。例如，在观察一辆汽车时，汽车内部和视点之间有一片玻璃；透过两片车窗玻璃，便可以看得见汽车后面的某些物体。

8.1.1 源因子和目标因子

在混合过程中，输入片原（即“源”）的颜色值将分两个阶段进行处理，与对应的当前所存储的像素（目标）颜色值进行合成。首先，要指定如何计算源因子和目标因子。这些因子是分别乘以源和目标中 R，G，B，A 各分量之后的 RGBA 四元组。然后，将两个 RGBA 四元组中各对应的分量相加。为了从数学上说明这一点，假定源和目标混合因子分别为（Sr, Sg, Sb, Sa) 和 (Dr, Dg, Db, Da),下标 s,d 表示源和目标的 RGBA 值。那么，混合后的 RGBA 值由下式给出：

$$(R_sS_r+R_dD_r,\ G_sS_g+G_dD_g,\ B_sS_b+B_dD_b,\ A_sS_a+A_dD_a)$$

该四元组的各分量值最终截取在区间[0,1]内。

现在，让我们来看一看源和目标的混合因子是如何生成的。此时需要利用函数 glBlendFunc()提供两个常数：一个指明如何计算源因子，另一个指明如何计算目标因子。为了使混合生效，还需要用下面的函数将其激活：

```
glEnable(GL_BLEND);
```

使用带有参数 GL_BLEND 的函数 glDisable()可以取消混合操作。还应注意，当混合被中止时，使用常数 GL_ONE（源）和 GL_ZERO（目标）的结果是一样的；这些值是缺省值。

---

void glBlendFunc(GLenum sfactor, GLenum dfactor)

此函数用于控制如何将已处理过的片原（源）的颜色值与存储在帧缓存（目标）中的片原颜色值合成到一起。参数 sfactor 说明如何计算源混合因子；dfactor 说明如何计算目标混合因子。这些参数的可能值列于表 8-1 中。假定混合因子在[0,1]范围内取值；源和目标颜色合成之后，其值就被截取在[0,1]中。

注意，在表 8-1 中，源和目标的 RGBA 值分别以下标 s 和 d 表示。RGBA 四元组的标量除法，表示用这些值去除以每个分量；类似地，四元组减法表示将分量逐个相减。“相关因子”这一列表示对应的常数是用来指定源混合因子还是指定目标混合因子。

表 8-1　源和目标混合因子

常数	相关因子	计算得到的混合因子
GL_ZERO	源或目标	(0, 0, 0, 0)
GL_ONE	源或目标	(1, 1, 1, 1)
GL_DST_COLOR	源	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	目标	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	源	$(1, 1, 1, 1)-(R_d, G_d, B_d, A_d)$
GL_ONE_MINUS_SRC_COLOR	目标	$(1, 1, 1, 1)-(R_s, G_s, B_s, A_s)$
GL_SRC_ALPHA	源或目标	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	源或目标	$(1, 1, 1, 1)-(A_s, A_s, A_s, A_s)$
GL_DST_ALPHA	源或目标	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	源或目标	$(1, 1, 1, 1)-(A_d, A_d, A_d, A_d)$
GL_SRC_ALPHA_SATURATE	源	$(f, f, f, 1)$; $f=\min(A_s, 1-A_d)$

8.1.2　混合的简单使用

源和目标因子的合成并不都是有意义的。多数应用程序都只用到一少部分合成。下面的内容描述了用于源和目标因子的特殊合成的典型用法。这些例子中，有些只用到了输入 alpha 值，因此，即使 alpha 值没有存储在帧缓存中，它们也可以照常工作。还应注意，实现这些效果的方法常常是多种多样的。

1）要将一幅图像的一半与另一幅图像的一半均匀地混合组成另一幅图像，可采用这样的办法：先将源因子设置成 GL_ONE，绘制第一幅图像；然后，将源因子和目标因子设置成 GL_SRC_ALPHA，以 alpha=0.5 绘制第二幅图像。如果希望合成后第一幅图像占 0.75，第二幅图像占 0.25，那么可以按前面的方法绘制第一幅图像，以 alpha=0.25 绘制第二幅图像即可，但是，这时的源因子为 GL_SRC_ALPHA，目标因子为 GL_ONE_MINUS_SRC_ALPHA。这两个因子代表了最常用的混合操作。

2）为了均匀地混合三幅不同的图像，先将目标因子设置成 GL_ONE，将源因子设置成 GL_SRC_ALPHA，以 alpha=0.3333333 绘制每一幅图像。利用这种技术，每一幅图像的亮

度只有原来图像的 1/3——若图像之间不遮盖重叠，将很明显也看到这一点。

3）假定在编写一段绘图程序的过程中，希望能够逐渐地增加画笔的颜色，使得每画一次都能在图像的当前内容混合进更多一点的颜色（比方说，每画一次都进行一次混合。在混合中，图像颜色占 90%，而画笔颜色占 10%）。要做到这一点，以 alpha=0.1，参数 GL_SRC_ALPHA（源）及 GL_ONE_MINUS_SRC_ALPHA（目标）来绘制画笔图像。（注意，可以沿画笔的宽度方向改变 alpha 的值，使得在画笔中能加上较多的颜色，而在边缘所加的颜色较少。参见反走样一节。）类似地，如果将其颜色设置为背景颜色，则可以实现橡皮擦的功能。

4）使用源或目标颜色的混合函数（GL_DST_COLOR 或 GL_ONE_MINUS_DST_COLOR 用于源因子，GL_SRC_COLOR 或 GL_ONE_MINUS_SRC_COLOR 用于目标因子）程序员便可以有效地、独立地调整各个颜色分量。这种操作相当于使用了一个简单的过滤器——例如，将红色分量乘以 0.8，绿色分量乘以 0.4，蓝色分量乘以 0.72，就可以模仿通过一个照相用滤光器来观察场景的效果，这种滤光器可以滤掉 20%的红光，60%的绿光和 28%的蓝光。

5）假定要将 3 幅半透明的表面合成一幅图画，这些表面彼此相互遮挡，所有表面都具有单色的背景。假定距离最远的表面可以透过其后 80%的颜色，接下来的表面可以透过 40%的颜色，距离最近的表面可以透过 90%的颜色。为了合成这幅图画，首先要以缺省的源和目标因子绘制背景，然后将混合因子改变为 GL_SRC_ALPHA（源）和 GL_ONE_MINUS_SRC_ALPHA（目标）。接下来，以 alpha=0.2 绘制最远的表面，以 alpha=0.6 绘制中间的表面，最后以 alpha=0.1 绘制最近的表面。

6）如果系统有 alpha 位平面，就可以对物体逐个进行渲染（包括 alpha 值），将它们读取回来，然后对经过了完整渲染的物体进行有趣地合成操作。注意，用于图像合成操作的物体可以有不同的来源——可以是 OpenGL 命令渲染的，可以是利用光线追踪等技术渲染的，也可以是用现存的图像通过扫描得到的。

7）程序员可以通过对图像的个别片原分配不同的 alpha 值，来创建非矩形光栅图像的效果。将每一个“不可见”的片原的 alpha 值设置为 0，每一个不透明片原的 alpha 值为 1.0。例如，可以将一个多边形绘制成树形，然后将树叶纹理粘贴到多边形上；如果 alpha=0，观察者看到的就是矩形纹理的一部分，而不是树的一部分。这种方法，有时叫作“贴画法”，比起创建三维的树要快得多。图 8-1 所示的例子就利用了这种技术。这棵树是一个单一的矩形，可以绕树干中心进行旋转，如轮廓线所示。因此，这棵树总是面对观察者的。关于混合纹理的详细信息，请参见“调整与混合”一节。

8.1.3 混合操作范例

下面给出了混合操作的一个例子——例 8-1，绘制了 4 个重叠的着色矩形，其 alpha 值均为 0.75，使得窗口的左下区域及右上区域均被覆盖了两次。在这两个区域中，颜色按不同

的次序利用 GL_SRC_ALPHA 及 GL_ONE_MINUS_SRC_ALPHA 的源和目标因子进行混合：在左下区域，将青色与初始颜色黄色混合；在右上区域，将黄色与初始颜色青色混合。其他区域仍按未经混合的颜色绘制。

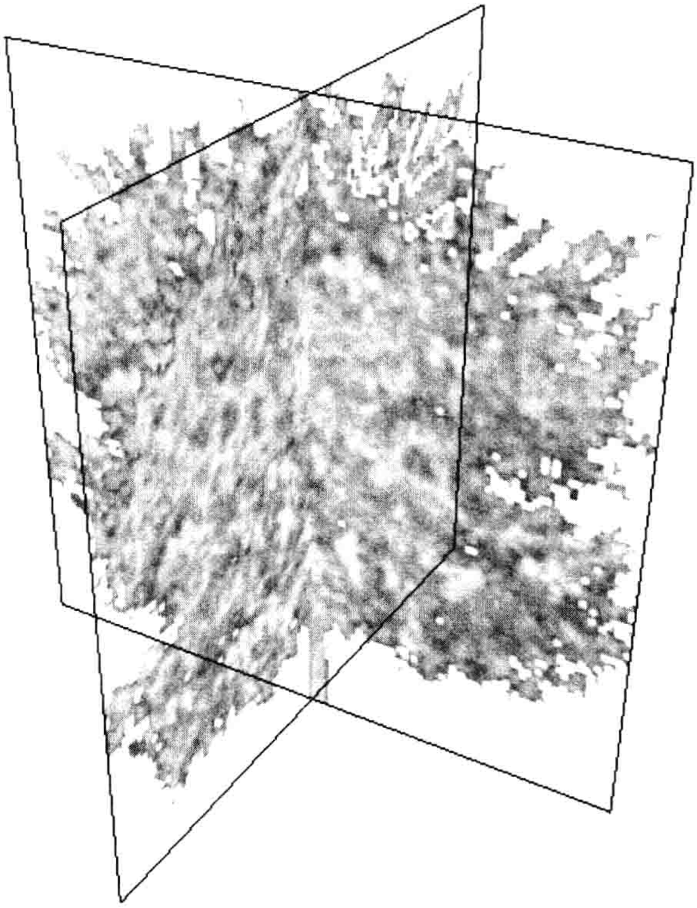


图 8-1　创建一幅非矩形光栅图像

例 8-1　一个混合操作的范例：alpha.c

```
#include "windows.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL\glaux.h>

void myinit(void)
{
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glShadeModel(GL_FLAT);
```

```
    glClearColor(0.0, 0.0, 0.0, 0.0);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor4f(1.0, 1.0, 0.0, 0.75);
    glRectf(0.0, 0.0, 0.5, 1.0);
    glColor4f(0.0, 1.0, 1.0, 0.75);
    glRectf(0.0, 0.0, 1.0, 0.5);

    /* 在右上角以相反次序绘制着色矩形 */

    glColor4f (0.0, 1.0, 1.0, 0.75);
    glRectf (0.5, 0.5, 1.0, 1.0);
    glColor4f (1.0, 1.0, 0.0, 0.75);
    glRectf (0.5, 0.5, 1.0, 1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D (0.0, 1.0, 0.0, 1.0*(GLfloat)h/(GLfloat)w);
    else
        gluOrtho2D (0.0, 1.0*(GLfloat)w/(GLfloat)h, 0.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
```

```
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 8-1");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return 0;
}
```

程序运行结果如图 8-2 所示。

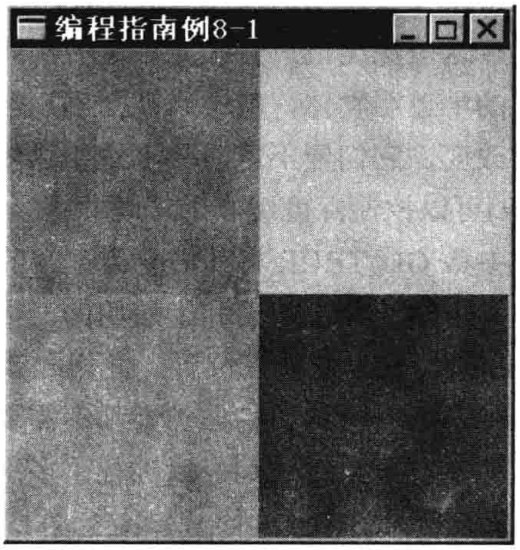


图 8-2　例 8-1 的运行结果

可以想象得到，矩形的绘制次序要影响最终的颜色。在左下区域，青色的矩形成为与黄色矩形混合的源片原，因为黄色矩形存储在帧缓存中，所以是目标片原。而在右上区域的情况正好相反。由于 alpha 值都是 0.75，因此源的实际混合因子是 0.75，目标的混合因子为 0.25。换句话说，源矩形有某种程度的半透明，它比目标矩形对最终颜色的影响更大。结果，左下区域就呈淡青色，右上区域呈淡黄色。经过计算可以发现，左下区域的 RGB 值为（0.25, 1.0, 0.75），而右上区域为（0.75, 1.0, 0.25）。

8.1.4　具有深度缓存的三维混合

和前面的例子一样，多边形的绘制次序也将在很大程度上影响混合结果。当绘制三维半透明物体时也可以得到不同的外观，这取决于程序员绘制多面体的次序——是从前面到后面，还是从后面到前面。在确定正确次序的同时，还要考虑深度缓存的影响。深度缓存（有时也叫 z 缓存）常常用于消除隐藏面（关于深度缓存的详细讨论，请参见第 11 章）。深度缓存追踪视点与占据了屏幕窗口某给定像素的这部分物体之间的距离；当该像素点的颜色改变时，只画出靠近视点的物体，较远的物体实际上被前面的物体遮挡住了。在这种情况下，深

度值是保存在深度缓存中的。利用这种方法，表面被隐藏（或遮盖）的部分就用不着画出来，因此也就用不着进行混合操作，这样一来，将使渲染速度大大地增加。

一般常需要对同一个场景中的不透明和半透明物体进行渲染，并且还希望利用深度缓存来消除不透明物体后面的隐藏面——如果不透明物体遮住了一个半透明物体或另一个不透明物体，就希望利用深度缓存来消除较远的物体。然而，如果半透明物体较近，则需要将它与后面的不透明物体进行混合。如果场景中的每个东西都是静止的，一般情况下是可以较为轻松地计算出多面体绘制的正确次序，但如果视点或物体是运动的，问题就变得复杂了。

对于这种情况，一个解决的办法就是将深度缓存激活，但在绘制半透明物体时，让深度缓存处于只读状态。首先以正常操作状态下的深度缓存绘制所有的不透明物体。然后，让深度缓存处于只读状态，以便保存这些深度值。在画完半透明物体后，这些深度值仍然没有变化，因此处于不透明物体后面的半透明物体，就不会被画出。但是，如果半透明物体靠近视点，由于深度缓存的值并没有改变，它们便不能清除不透明物体而是要与不透明物体混合在一起。利用函数 glDepthMask()可以控制深度缓存的读写状态。如果参数为 GL_FALSE，则深度缓存为只读状态，反之，参数 GL_TRUE 将其恢复为正常的可写状态。

例 8-2 例示了如何利用这种方法绘制不透明和半透明的三维物体。在程序中，按下鼠标左键时调用函数 toggleviewpoint()，来改变视点位置，并由此改变不透明圆环和半透明圆柱的次序。注意，只有在帧缓存中，所有的透明物体像素都不需要多次绘制时，这种方法才较准确。如果半透明物体相互重叠，将导致对个别像素进行了多次混合渲染，此时这种方法仅可作为一种近似。

例 8-2 三维混合

```
#include "windows.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL\glaux.h>

void myinit(void)
{
GLfloat mat_ambient[] = { 0.0, 0.0, 0.0, 0.15 };
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 0.15 };
GLfloat mat_shininess[] = { 15.0 };
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
glEnable(GL_LIGHTING);
```

```
glEnable(GL_LIGHT0);
glDepthFunc(GL_LEQUAL);
glEnable(GL_DEPTH_TEST);
}

GLboolean eyePosition = GL_FALSE;

void CALLBACK toggleEye(AUX_EVENTREC *event)
{
    if (eyePosition)
        eyePosition = GL_FALSE;
    else
        eyePosition = GL_TRUE;
}

void CALLBACK display(void)
{
    GLfloat position[] = { 0.0, 0.0, 1.0, 1.0 };
    GLfloat mat_torus[] = { 0.75, 0.75, 0.0, 1.0 };
    GLfloat mat_cylinder[] = { 0.0, 0.75, 0.75, 0.15 };
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glPushMatrix();

    if (eyePosition)
         gluLookAt(0.0, 0.0, 9.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    else
         gluLookAt(0.0, 0.0, -9.0, 0.0, 0.0, 0.0, 0.0,1.0, 0.0);

    glPushMatrix();
    glTranslatef(0.0, 0.0, 1.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_torus);
    auxSolidTorus(0.275, 0.85);
    glPopMatrix();
    glEnable(GL_BLEND);
```

```
    glDepthMask(GL_FALSE);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_cylinder);
    glTranslatef(0.0, 0.0, -1.0);
    auxSolidCylinder(1.0, 2.0);
    glDepthMask(GL_TRUE);
    glDisable(GL_BLEND);
    glPopMatrix();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
        gluPerspective(30.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
}

int main(int argc, char** argv)
{
   auxInitDisplayMode(AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
   auxInitPosition(0, 0, 300, 300);
   auxInitWindow("编程指南例 8-2");
   auxMouseFunc(AUX_LEFTBUTTON, AUX_MOUSEDOWN, toggleEye);
   myinit();
   auxReshapeFunc(myReshape);
   auxMainLoop(display);
    return 0;
}
```

程序运行结果如图 8-3 所示。

图 8-3　例 8-2 的运行结果

8.2　反走样

读者可能已经注意到，在某些 OpenGL 图像中，直线尤其是近乎水平或垂直的线条，看起来是锯齿形的。出现这种变形的原因是，理想的直线在计算机图形中总是用一系列像素来近似的，而这些像素又必须位于像素栅格内。这种参差不齐的状态称之为走样现象。本节描述了反走样。图 8-4 显示了两条相交直线，左图中两条直线都有走样，在右图中使用了反走样技术。为了显示这种效果，这两幅图片已经被放大了。

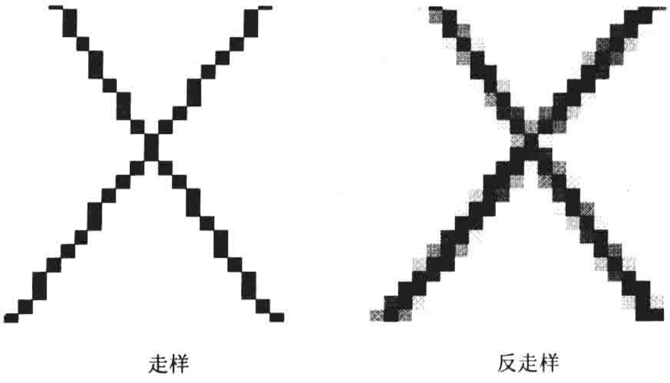


图 8-4　直线的走样和反走样

图 8-4 例示了斜线中的一个像素宽度如何去覆盖比正常情况下更多的像素区域。事实上，在消除锯齿现象时，OpenGL 要基于屏幕像素块所覆盖的区域，对每一块片原的覆盖范围值进行计算。

图 8-5 显示了直线的覆盖范围值。在 RGBA 模式中，OpenGL 要用该值与片原的 alpha 值相乘。然后，利用结果 alpha 值将片原与帧缓存中的对应像素混合。在颜色索引模式下，

OpenGL 基于片原的覆盖值（0000 表示没有覆盖，1111 表示全部覆盖）对 4 位颜色索引进行设置。加载颜色图，并适当地通过它来利用覆盖信息，这些都是必须由程序员负责完成的事情。

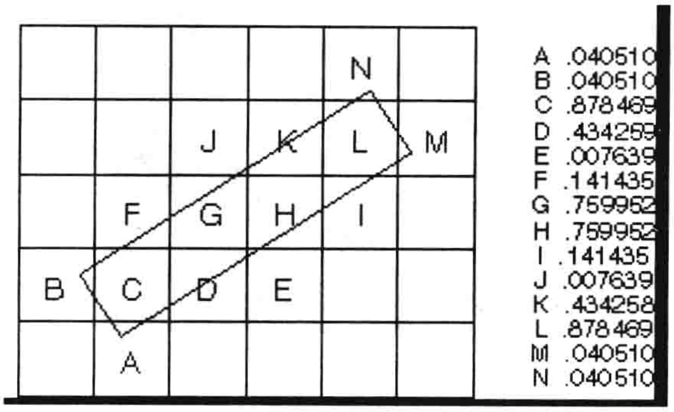


图 8-5 确定覆盖值

计算覆盖值常常是比较复杂的，一般难以进行说明，而且实际上，在 OpenGL 的不同实现中也多少存在一些差别。可以利用 glHint()命令试着去控制和调整图像质量与速度这对矛盾，但并非所有的实现都会采用这一命令中给出的提示。

---

void glHint(GLenum target, GLenum hint);

此函数控制 OpenGL 的某些行为。参数 target 表示要控制何种行为，它的可能取值列于表 8-2 中。参数 hint 可以是 GL_FASTEST，表示将使用最大效率选项；也可以是 GL_NICEST，表示使用质量最好的选项；或者是 GL_DONT_CARE，表示没有首选项。对提示的解释是和 OpenGL 具体的实现相关的；在某种实现中可能完全忽略它们。

表 8-2 用于函数 glHint()的参数及其含义

参数	含义
GL_POINT_SMOOTH_HINT, GL_LINE_SMOOTH_HINT, GL_POLYGON_SMOOTH_HINT	在消除锯齿现象的操作过程中，指定直线，点，或者多边形的期望采样品质
GL_FOG_HINT	确定是对每一个像素计算雾（GL_NICEST），还是对每一个顶点计算雾(GL_FASTEST)
GL_PERSPECTIVE_CORRECTION_HINT	指定颜色和纹理坐标插值的期望质量

关于相关主题更详细的信息可以在本书其他部分得到参考，如采样的细节请参见“反走样”一节，雾的细节参见“雾”一节。参数 GL_PERSPECTIVE_CORRECTION_HINT 指的

是如何在图元中对颜色值和纹理坐标进行插值：或者是在屏幕空间进行插值（计算相当简单），或者以透视修正的方式进行插值（需要的计算量较大）。通常，系统对颜色进行线性插值，因为这种插值结果在视觉上是可以接受的；然而，对于纹理，多数情况下需要进行透视修正插值，这样，在视觉上才能符合要求。因此，一个具体的实现可以有选择地使用这个参数，以便对所用的插值方法进行控制。“透视投影”和“颜色”已分别在第 4 章和第 6 章中进行了讨论，“纹理映射”将在第 10 章中讨论。

8.2.1　点和线的反走样

为了消除点和线的走样，需要用适当的参数，如 GL_POINT_SMOOTH 或 GL_LINE_SMOOTH，来调用函数 glEnable()，以激活消除锯齿现象的功能。也可以用函数 glHint()提示所需要的品质（记住，可以设置点的大小或线的宽度，也可设置为点画线。详情请参见第 3 章）。然后，根据所处的是 RGBA 模式还是颜色索引模式，遵照后面部分所描述的过程进行操作。

1. RGBA 模式下

在 RGBA 模式中，需要将混合操作激活。最有可能用到的混合因子是 GL_SRC_ALPHA（源）和 GL_ONE_MINUS_SRC_ALPHA（目标）。也可以将目标因子定为 GL_ONE，使直线的相交处更亮。现在就可以着手绘制需要进行反走样的点或线了。如果 alpha 值较高，反走样效果将是非常明显的。

记住，由于执行了混合操作，正如具有深度缓存的三维混合一节中描述的那样，需要考虑渲染的次序。然而，在大多数情况下，如果没有大的不利影响，次序是可以忽略的。例 8-3 先对反走样所必需的模式进行初始化，然后绘制一个 20 面体的线框图。注意，在本例中，并没有激活深度缓存。

例 8-3　一个经过反走样操作的 20 面体的线框图

```
#include "windows.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void)
{
    GLfloat values[2];
    glGetFloatv(GL_LINE_WIDTH_GRANULARITY, values);
    printf("GL_LINE_WIDTH_GRANULARITY value is %3.1f\n",
          values[0]);
```

```
    glGetFloatv(GL_LINE_WIDTH_RANGE, values);
    printf("GL_LINE_WIDTH_RANGE values are %3.1f %3.1f\n",
            values[0], values[1]);
    glEnable(GL_LINE_SMOOTH);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glHint(GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
    glLineWidth(1.5);
    glShadeModel(GL_FLAT);
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor4f(1.0, 1.0, 1.0, 1.0);
    auxWireIcosahedron(1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (GLfloat) w/(GLfloat) h, 3.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -4.0);
}

int main(int argc, char** argv)
{
```

```
    auxInitDisplayMode(AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition(0, 0, 300, 300);
    auxInitWindow("编程指南例 8-3");
    myinit();
    auxReshapeFunc(myReshape);
    auxMainLoop(display);
    return 0;
}
```

程序运行结果如图 8-6 所示。

图 8-6　例 8-3 的运行结果

2. 颜色索引模式下

在颜色索引模式中，消除锯齿现象的关键部分就是加载并使用颜色图。由于颜色索引的后 4 位表示的是覆盖范围值，因此需要用背景颜色与物体颜色形成的颜色梯度加载 16 个相邻的索引（梯度必须以 16 为倍数的索引值开始）。然后，将梯度中 16 种颜色的第一种颜色缓存清空，并利用梯度中的颜色绘制点和线。例 8-4 演示了如何构造颜色梯度，以便在颜色索引模式下绘制已经消除锯齿现象的 20 面体的线框图。在本例中，颜色梯度以索引 32 开始，并包含了灰度阴影。

例 8-4　在颜色索引模式下消除锯齿现象

```
#include "windows.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL\glaux.h>
#define RAMPSIZE 16
#define RAMPSTART 32
```

```
void myinit(void)
{
    int i;
    for (i = 0; i < RAMPSIZE; i++)
      {
        GLfloat shade;
        shade = (GLfloat) i/(GLfloat) RAMPSIZE;
        auxSetOneColor(RAMPSTART+(GLint)i, shade, shade, shade);
      }

    glEnable (GL_LINE_SMOOTH);
    glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
    glLineWidth (1.5);
    glClearIndex ((GLfloat) RAMPSTART);
    glShadeModel(GL_FLAT);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glIndexi(RAMPSTART);
    auxWireIcosahedron(1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLfloat) w/(GLfloat) h, 3.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
```

```
    glTranslatef (0.0, 0.0, -4.0);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_INDEX | AUX_DEPTH);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 8-4");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return 0;
}
```

程序运行结果如图 8-7 所示。

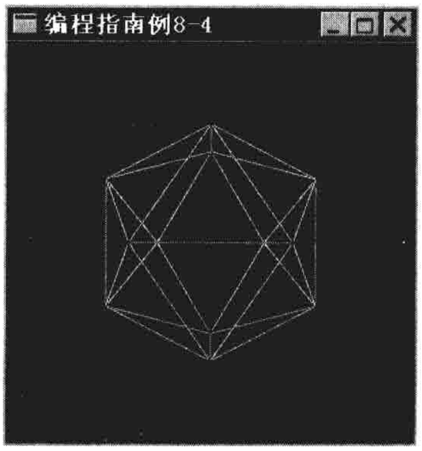


图 8-7　例 8-4 的运行结果

由于颜色梯度是以背景颜色和物体颜色的差值来构造的，因此只有当消除锯齿现象的物体画到相应背景上时，才会显现为正确图像。如果消除锯齿现象的物体被画在其他物体上，对此未加考虑，仍然用原来的颜色梯度，在相交的地方物体的颜色就会出现异常。为了得到最佳结果，要利用深度缓存以确保像素颜色与“最近”的物体相对应。在 RGBA 模式下，两个物体的颜色将被混合到一起，使结果看起来更自然。因此，对消除了锯齿现象的点和直线组成的场景进行渲染时，通常用不着使用深度缓存。

在带有深度缓存的三维混合一节中描述的办法也可以用于将没有锯齿的点和直线与有锯齿及深度缓存的多边形混合到一起。要实现上述过程，需要首先绘制多边形，然后使深度缓存的状态为只读，再绘制点和直线。点和直线在相交处非常精细，但它们会被近处的多边形遮盖。

试一试

利用前面的程序，如组合变换实例一节中描述的机器人臂或太阳系程序，绘制消除锯齿现象的物体的线框图。在 RGBA 模式和颜色索引模式中都试一试。也可以试一试不同宽度的直线或不同大小的点，看看其效果如何。

8.2.2 多边形的反走样

填充多边形的边的锯齿现象的消除，可以采取类似于点或直线锯齿现象消除的方法。当不同的多边形有互相遮盖的部分时，需要适当地混合这些颜色值。消除整个场景的锯齿现象可以利用本节描述的方法，也可以利用累积缓存。在第 11 章描述的累积缓存，使用虽然更加容易，但计算量大大增加，因此速度较慢。

注意：如果通过把 glPolygonMode()的参数设置为 GL_POINT 或 GL_LINE，将多边形绘成顶点处点的形式或轮廓线的形式，就需要应用前面描述的点或直线的锯齿现象消除方法。本节的剩下部分着重讨论 GL_FILL 方式下的填充多边形反走样。

理论上，在 RGBA 模式或颜色索引模式中，均可以进行消除多边形锯齿现象的操作。然而，物体的相交对消除多边形锯齿现象的影响比对点或直线的影响更大，因此，对渲染次序和混合精度都提出了更严格的要求。实际上严格达到了这样的程度：如果要消除多个多边形的锯齿现象，首先必须从前到后对多边形进行排序，然后以源因子为 GL_SRC_ALPHA_SATURATE，目标因子为 GL_ONE 调用函数 glBlendFunc()。因此，在颜色索引模式中要消除多边形的锯齿现象通常是不实际的。

为了在 RGBA 模式下消除多边形的锯齿现象，应使用 alpha 值来表示多边形边缘的覆盖值，还需要以 GL_POLYGON_SMOOTH 为参数调用函数 glEnable()，激活多边形的“反走样”功能。而后便可以根据其覆盖值来分配多边形边缘像素的 alpha 值，尽管它们是一些经过锯齿现象消除的直线。另外也可以根据自己的意愿给 GL_POLYGON_SMOOTH_HINT 提供一个 alpha 值。

下面，需要适当地对重叠的边进行混合。首先，关掉深度缓存，这样就可以对如何绘制重叠像素进行控制。然后将混合因子设置成 GL_SRC_ALPHA_SATURATE（源）和 GL_ONE（目标）。利用这个专用的混合函数，最终的颜色就变成目标颜色和变换后的源颜色的相加；变换因子是输入源的 alpha 值和负的目标 alpha 值中较小的一个。这就意味着，对于有较大 alpha 值的像素，后面渲染的像素对最终颜色将没有太大影响，因为，这一像素值与目标 alpha 值的差几乎为 0。利用这种方法，多边形边缘的像素最终就可以与稍后绘制的其他多边形颜色进行混合。最后，需要在绘制前对场景中所有多边形按从前到后的顺序进行排序。

例 8-5 例示了如何消除填充多边形的锯齿现象；按下鼠标左键可以开关锯齿现象消除的功能。注意，背面多边形已被去掉，而且，在绘制之前，颜色缓存中的 alpha 值已被清 0（对于这种技术，必须在颜色缓存中存储 alpha 值才能正常工作）。

例 8-5　填充多边形的锯齿现象消除

```
#include "windows.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL\glaux.h>

GLboolean polySmooth;

void myinit(void)
{
    GLfloat mat_ambient[] = { 0.0, 0.0, 0.0, 1.00 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.00 };
    GLfloat mat_shininess[] = { 15.0 };
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glEnable (GL_LIGHTING);
    glEnable (GL_LIGHT0);
    glEnable (GL_BLEND);
    glCullFace (GL_BACK);
    glEnable (GL_CULL_FACE);
    glEnable (GL_POLYGON_SMOOTH);
    polySmooth = GL_TRUE;
    glClearColor (0.0, 0.0, 0.0, 0.0);
}

void CALLBACK toggleSmooth (AUX_EVENTREC *event)
{
    if (polySmooth)
       {
          polySmooth = GL_FALSE;
          glDisable (GL_BLEND);
          glDisable (GL_POLYGON_SMOOTH);
          glEnable (GL_DEPTH_TEST);
```

```
        }
    else
      {
          polySmooth = GL_TRUE;
          glEnable (GL_BLEND);
          glEnable (GL_POLYGON_SMOOTH);
          glDisable (GL_DEPTH_TEST);
    }
}

void CALLBACK display(void)
{
    GLfloat position[] = { 0.0, 0.0, 1.0, 0.0 };
    GLfloat mat_cube1[] = { 0.75, 0.75, 0.0, 1.0 };
    GLfloat mat_cube2[] = { 0.0, 0.75, 0.75, 1.0 };

    if (polySmooth)
         glClear (GL_COLOR_BUFFER_BIT);
    else
         glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix ();
    glTranslatef (0.0, 0.0, -8.0);
    glLightfv (GL_LIGHT0, GL_POSITION, position);
    glBlendFunc (GL_SRC_ALPHA_SATURATE, GL_ONE);
    glPushMatrix ();
    glRotatef (30.0, 1.0, 0.0, 0.0);
    glRotatef (60.0, 0.0, 1.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_cube1);
    auxSolidCube (1.0, 1.0, 1.0);
    glPopMatrix ();
    glTranslatef (0.0, 0.0, -2.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_cube2);
    glRotatef (30.0, 0.0, 1.0, 0.0);
    glRotatef (60.0, 1.0, 0.0, 0.0);
```

```
    auxSolidCube (1.0);
    glPopMatrix ();
    glFlush ();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(30.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
   auxInitPosition (0, 0, 200, 200);
   auxInitWindow ("编程指南例 8-5");
   auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, toggleSmooth);
   myinit();
   auxReshapeFunc (myReshape);
   auxMainLoop(display);
    return 0;
}
```

程序运行结果如图 8-8 所示。

图 8-8　例 8-5 的运行结果

8.3 雾

计算机图像有时轮廓太分明、太尖锐，因而显得不真实。消除锯齿现象通过对边缘进行平滑使物体看起来更真实。另外，可以通过增加雾的方法来制作更自然的完整图像。这里雾是一个通用的术语，它描述了类似于大气效果的各种形态；它可以用来模拟迷雾、薄雾、烟尘、以及浊气。在可视化仿真中，雾是必须的，因为可视化仿真需要适当地对有限的能见度进行模拟。所以雾的操作常用于飞行模拟器的显示中。

当雾被激活时，离视点较远的物体就淡化到雾色之中。雾的密度是可以进行控制的，它与雾的颜色一样，决定了物体淡化的速度。在 RGBA 模式及颜色索引模式中，雾都是可用的，只是在计算上稍有不同。因为雾的使用是在矩阵变换、光照、纹理等处理过程之后，因此对变换过的、有光照的、有纹理的物体都会产生影响。注意，在大多数的仿真程序中，雾都能改进程序执行性能，因为此时由于雾重而不可见的物体可以不用画出。

8.3.1 雾的使用

雾的使用是非常简单的，以 GL_FOG 为参数调用函数 glEnable()就可以激活雾，雾的颜色和进行密度控制的方程则可以利用函数 glFog*()进行选择。如果需要的话，还可以用函数 glHint()为 GL_FOG_HINT 提供一个值，如表 8-2 所描述的那样。在例 8-6 中，绘制了 5 个红色的茶壶，每个茶壶与视点的距离不同。按下鼠标左键，可以在三个不同的雾方程中进行选择。雾的方程将在下一节中进行描述。

例 8-6 RGBA 模式下笼罩在雾中的五个茶壶

```
#include "windows.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL\glaux.h>

GLint fogMode;

void CALLBACK cycleFog (AUX_EVENTREC *event)
{
    if (fogMode == GL_EXP)
    {
      fogMode = GL_EXP2;
```

```
        printf ("Fog mode is GL_EXP2\n");
    }
    else if (fogMode == GL_EXP2)
    {
        fogMode = GL_LINEAR;
        printf ("Fog mode is GL_LINEAR\n");
        glFogf (GL_FOG_START, 1.0);
        glFogf (GL_FOG_END, 5.0);
    }
    else if (fogMode == GL_LINEAR)
    {
        fogMode = GL_EXP;
        printf ("Fog mode is GL_EXP\n");
    }
    glFogi (GL_FOG_MODE, fogMode);
}

void myinit(void)
{
    GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };
    GLfloat local_view[] = { 0.0 };
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, local_view);
    glFrontFace (GL_CW);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    glEnable(GL_FOG);

      {
          GLfloat density;
          GLfloat fogColor[4] = {0.5, 0.5, 0.5, 1.0};
```

```
        fogMode = GL_EXP;
        glFogi (GL_FOG_MODE, fogMode);
        glFogfv (GL_FOG_COLOR, fogColor);
        glFogf (GL_FOG_DENSITY, 0.35);
        glHint (GL_FOG_HINT, GL_DONT_CARE);
        glClearColor(0.5, 0.5, 0.5, 1.0);
    }
}

void CALLBACK renderRedTeapot (GLfloat x, GLfloat y, GLfloat z)
{
    float mat[3];
    glPushMatrix();
    glTranslatef (x, y, z);
    mat[0] = 0.1745;
    mat[1] = 0.01175;
    mat[2] = 0.01175;
    glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
    mat[0] = 0.61424;
    mat[1] = 0.04136;
    mat[2] = 0.04136;
    glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
    mat[0] = 0.727811;
    mat[1] = 0.626959;
    mat[2] = 0.626959;
    glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
    glMaterialf (GL_FRONT, GL_SHININESS, 0.6*128.0);
    auxSolidTeapot(1.0);
    glPopMatrix();
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    renderRedTeapot (-4.0, -0.5, -1.0);
```

```
    renderRedTeapot (-2.0, -0.5, -2.0);
    renderRedTeapot (0.0, -0.5, -3.0);
    renderRedTeapot (2.0, -0.5, -4.0);
    renderRedTeapot (4.0, -0.5, -5.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= (h*3))
        glOrtho (-6.0, 6.0, -2.0*((GLfloat) h*3)/(GLfloat) w,
                 2.0*((GLfloat) h*3)/(GLfloat) w, 0.0, 10.0);
    else
        glOrtho (-6.0*(GLfloat) w/((GLfloat) h*3),
                 6.0*(GLfloat) w/((GLfloat) h*3), -2.0, 2.0, 0.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 450, 150);
    auxInitWindow ("编程指南 8-6");
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, cycleFog);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return 0;
}
```

运行结果如图 8-9 所示。

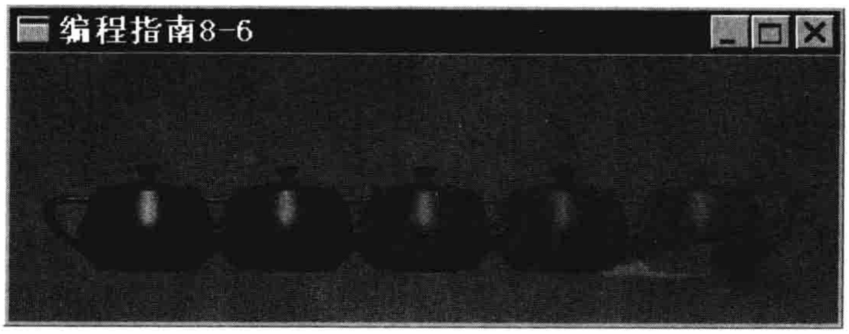


图 8-9 例 8-6 的运行结果

8.3.2 雾的方程

利用雾的混合因子，将雾的颜色与输入片原的颜色相混合。雾的混合因子 f 是根据下面三个方程之一来进行计算的，其取值限制在[0,1]之内。

$$
\begin{cases}
f = e^{-(density \cdot z)} & (GL_EXP) \\
f = e^{-(density \cdot z)^2} & (GL_EXP2) \\
f = \dfrac{end - z}{end - start} & (GL_LINEAR)
\end{cases}
$$

其中，z 为视点与片原中心的人眼坐标之间的距离。密度、起点及终点的值是由 glFog*() 确定的。根据 RGBA 模式和颜色索引模式的不同，f 因子的使用是不同的。这一点将在下面解释。

---

void glFog{if}{v}(GLenum pname, TYPE param);

此函数设置用于计算雾的参数和函数。如果 pname 为 GL_FOG_MODE，那么 param 可以是 GL_EXP (缺省), GL_EXP2 或 GL_LINEAR。如果 pname 为 GL_FOG_DENSITY, GL_FOG_START 或 GL_FOG_END, 那么，param 就是（或者在向量形式下，指向）方程中的密度、起点或终点值（缺省值分别为 1，0，1）。在 RGBA 模式中，pname 的值可以是 GL_FOG_COLOR，这种情况下，param 是指向用于确定 RGBA 颜色的 4 个值的指针。在颜色索引模式中，pname 对应的值是 GL_FOG_INDEX，因此 param 是用于确定雾的颜色索引的一个值。

图 8-10 绘出了变参数的雾密度方程。利用线性雾可以实现深度提示效果，如彩图 2 所示。

1. RGBA 模式下的雾

在 RGBA 模式中，雾因子将应用如下，以便计算雾的最终颜色：

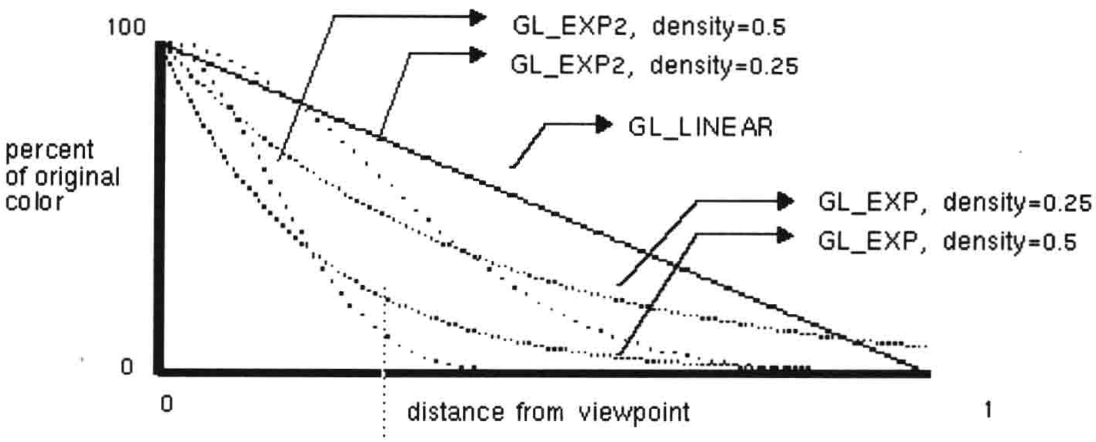


图 8-10 雾密度方程

$$C = f\,C_i + (1 - f)\,C_f$$

其中，C_i 表示输入片原的 RGBA 值，C_f 表示由 GL_FOG_COLOR 分配的雾的颜色值。

2. 颜色索引模式下的雾

在颜色索引模式中，雾的最终颜色索引按下式计算：

$$I = I_i + (1 - f)\,I_f$$

其中，I_i 是输入片原的颜色索引，I_f 相当于由 GL_FOG_INDEX 所指定的雾的颜色索引。

要在颜色索引模式下使用雾，必须在颜色梯度中加载适当的值。梯度中第一个颜色为没有雾时的物体颜色，最后一个颜色是完全浸在雾中的物体颜色。有时需要使用 glClearIndex() 来初始化背景颜色索引，以便与颜色梯度中的最后颜色相对应。通过这种方法，所有浸在雾中的物体将与背景混合到一起。类似地，在绘制物体之前，应该用梯度中第一个颜色（无雾的颜色）的索引作为参数调用函数 glIndex*()。最后，为了将雾作用于场景中具有不同颜色的物体，就需要创建多个颜色梯度，并且，要在绘制每个物体之前，调用函数 glIndex*()，将当前颜色索引设置成每个颜色梯度的初始值。例 8-7 例示了如何对适当的条件进行初始化，然后再在颜色索引模式中使用雾。

例 8-7 在颜色索引模式下使用雾

```
#include "windows.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

#define NUMCOLORS 32
#define RAMPSTART 16
```

```
void myinit(void)
{
    int i;
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    for (i = 0; i < NUMCOLORS; i++)
    {
     GLfloat shade;
     shade = (GLfloat) (NUMCOLORS-i)/(GLfloat) NUMCOLORS;
     auxSetOneColor (16 + i, shade, shade, shade);
    }
    glEnable(GL_FOG);
    glFogi (GL_FOG_MODE, GL_LINEAR);
    glFogi (GL_FOG_INDEX, NUMCOLORS);
    glFogf (GL_FOG_START, 0.0);
    glFogf (GL_FOG_END, 4.0);
    glHint (GL_FOG_HINT, GL_NICEST);
    glClearIndex((GLfloat) (NUMCOLORS+RAMPSTART-1));
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
    glTranslatef (-1.0, -1.0, -1.0);
    glRotatef (-90.0, 1.0, 0.0, 0.0);
    glIndexi (RAMPSTART);
    auxSolidCone(1.0, 2.0);
    glPopMatrix ();
    glPushMatrix ();
    glTranslatef (0.0, -1.0, -2.25);
    glRotatef (-90.0, 1.0, 0.0, 0.0);
    glIndexi (RAMPSTART);
    auxSolidCone(1.0, 2.0);
    glPopMatrix ();    glPushMatrix ();
```

```
    glTranslatef (1.0, -1.0, -3.5);
    glRotatef (-90.0, 1.0, 0.0, 0.0);
    glIndexi (RAMPSTART);
    auxSolidCone(1.0, 2.0);
    glPopMatrix ();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        if (w <= h)
            glOrtho (-2.0, 2.0, -2.0*(GLfloat)h/(GLfloat)w,
                2.0*(GLfloat)h/(GLfloat)w, 0.0, 10.0);
        else
            glOrtho (-2.0*(GLfloat)w/(GLfloat)h,
                2.0*(GLfloat)w/(GLfloat)h, -2.0, 2.0, 0.0, 10.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity ();
}

int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_INDEX | AUX_DEPTH);
   auxInitPosition (0, 0, 300, 300);
   auxInitWindow ("编程指南例 8-7");
   myinit();
   auxReshapeFunc (myReshape);
   auxMainLoop(display);
    return 0;
}
```

程序运行结果如图 8-11 所示。

图 8-11　例 8-7 的运行结果

第 9 章　绘制像素、位图、字体和图像

本章主题

在本章以前所讨论的主要是对几何数据——点、线和多边形的渲染。OpenGL 能够进行渲染的另两类重要的对象是：

1）位图，主要应用于各字体中的字符。

2）图像数据，可通过扫描或计算得到。

位图和图像数据均采用像素的矩形阵列形式来表示。二者的不同之处在于，位图只有每一像素的一位信息，而图像数据则包含每一像素的多个信息（比如完整的 R，G，B 和 alpha 颜色分量）。另外一点不同是，位图可以用于遮盖其他图像，类似于掩码，而图像数据绘制时，则是简单地覆盖或混合先前存在的数据。

本章将描述如何将像素数据（包括位图和图像）从帧缓冲器读取到处理器内存中（或其反过程），以及如何实现像素数据在不同缓存之间、或在同一缓存的不同位置之间进行拷贝。本章包括以下几个主要部分：

- 位图和字体描述定位和绘制位图化数据的命令。这些数据可以用来描述一种字体。
- 图像一节说明了关于像素数据的读取和拷贝的基本知识。另外阐述了图像在写入帧缓存时如何进行缩放。
- 像素的存储，变换和映射，详细描述了像素数据在内存中的存储格式，以及在读写内存时如何进行变换。

在多数情况下，只需要进行一些简单的像素操作，所以前两部分的内容便可以满足基本的应用要求。然而，实际的像素操作是很复杂的，比如内存中像素数据的存储就有许多种方式，而进行帧缓冲器读写时像素的变换又有许多种选择。关于像素操作的细节将在本章的第三部分讨论；这一部分可以根据实际应用的需要选读。

9.1　位图和字体

位图是 0 和 1 组成的矩阵，通过绘图对相应的矩形窗口部分进行屏蔽。假设要绘制红色的位图，位图中为 1 的对应像素均由红色像素替代（或与红色的像素点混合，这取决于哪一个片原操作有效；参见片原的测试与操作）。位图中为 0 处，对应的像素内容不受影响。位图主要用于在屏幕上显示字符。

OpenGL 对字符串的绘制和字体的操作仅仅提供了最底层的支持。glRasterPos*() 和

glBitmap()命令可以实现在屏幕上对一个位图的定位和绘制。而且，通过显示列表机制，可以用一系列的字符代码来索引代表这些字符的一系列位图。关于显示列表的内容，请参见第 5 章。对于位图、字体和字符串的其他操作，则必须由程序员自己编写程序来完成。

在例 9-1 中，将字符 F 在屏幕上画了三次。这个例子说明了作为位图的字符 F 和其相应的位图数据，如图 9-1 所示。

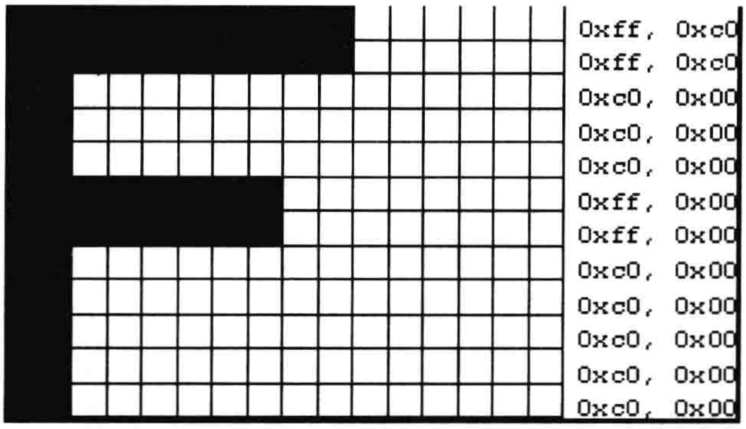


图 9-1 位图化的字符 F 及其数据

例 9-1 绘制位图字符

```
#include <windows.h>

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/aux.h>

GLubyte rasters[24] =
{
      0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
      0xc0, 0x00, 0xff, 0x00, 0xff, 0x00, 0xc0, 0x00,
      0xc0, 0x00, 0xc0, 0x00, 0xff, 0xc0, 0xff, 0xc0
};

void myinit(void)
{
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glClearColor(0.0, 0.0, 0.0, 0.0);
}
```

```
void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glRasterPos2i (20.5, 20.5);
    glBitmap(10, 12, 0.0, 0.0, 12.0, 0.0, rasters);
    glBitmap(10, 12, 0.0, 0.0, 12.0, 0.0, rasters);
    glBitmap(10, 12, 0.0, 0.0, 12.0, 0.0, rasters);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, w, 0, h, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(void/*int argc, char** argv*/)
{
   auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
   auxInitPosition(0, 0, 300, 300);
   auxInitWindow("编程指南例 9-1");
   myinit();
   auxReshapeFunc(myReshape);
   auxMainLoop(display);
    return 0;
}
```

程序运行结果如图 9-2 所示。

在图 9-1 中，可以注意到字符 F 的可见部分最多占 10 个位宽度。位图数据的存储是以 8 位为单位按块进行的，但实际显示的位图宽度并不一定是 8 的倍数。组成位图的各位是从左下角开始绘制的，即最先画出底部一行，然后是向上一行，依次绘出整个位图。从代码中可以看出，位图在内存当中的存储是按照这样的顺序：光栅阵列是从字符 F 底部两行 0xc0,

0x00, 0xc0, 0x00 开始，一直到顶部两行 0xff, 0xc0, 0xff, 0xc0 结束。

图 9-2 例 9-1 的运行结果

在例 9-1 中需要注意两个命令：glRasterPos2i() 和 glBitmap(); 下一部分将详细讨论这两个命令。这里暂不讨论对命令 glPixelStorei()的调用，该命令描述了位图在计算机内存中存储的格式。这一点，将在“像素存储模式的控制”这一部分中进行讨论。

9.1.1 当前光栅位置

当前光栅位置是下一个将要绘制的位图或图像的原点。在字符 F 的例子当中，光栅位置是用 glRasterPos*()命令设置在（20, 20）处，也就是所绘制的字符 F 的左下角。

```
glRasterPos2i(20, 20);
```

---

void glRasterPos{234}{sifd}{v}(TYPE x, TYPE y, TYPE z, TYPE w);

此函数设置当前的光栅位置。参数 x, y, z 和 w 指定了光栅位置的坐标。如果使用 glRasterPos2*()的形式，z 将被隐式地设置为 0，而 w 则被隐式地设置为 1；glRasterPos3*()与此类似，w 也被设置为 1。

由光栅位置坐标经由模型视图矩阵和投影矩阵变换的屏幕坐标，与 glVertex*()命令提供的坐标是相同的。在变换之后，可能在屏幕窗口定义了一个有效点，也可能由于变换坐标处在视区之外最终被剪裁掉。如果变换点被剪裁掉了，那么当前光栅位置就是无效的。

为了获得当前光栅位置，可以使用查询命令 glGetFloatv()。第一个参数取 GL_CURRENT_RASTER_POSITION。第二个参数是一个指针，指向一个经过分配可以浮点数形式存储（x, y, z, w）的数组。确定当前的光栅位置是否有效，可以使用命令 glGetBooleanv()，并将 GL_CURRENT_RASTER_POSITION_VALID 作为其第一个参数。

9.1.2 绘制位图

在设置了期望的光栅位置之后，就可以调用 glBitmap()来绘制位图了。

---

void glBitmap(GLsizei width, GLsizei height, GLfloat x_{bo}, GLfloat y_{bo}, GLfloat x_{bi}, GLfloat y_{bi}, const GLubyte *bitmap);

此函数绘制由 bitmap 所指定的位图，这里 bitmap 是指向位图图像的指针。位图的原点置于最新确定的光栅位置。如果当前光栅位置是无效的，则是空操作，光栅位置仍保持为无效。width 和 height 两个参数指定了以像素为单位的位图的宽和高。尽管每一个无符号字符型数据是占 8 位空间的，但 width 并不一定是 8 的倍数。（在字符 F 的例子中，10 位以外的各位数据中是否有无效位并没有什么关系；以宽度 10 来调用 glBitmap()的，每行只渲染了 10 位。）使用 x_{bo} 和 y_{bo} 来定义位图的原点（正值使原点向上，向右移动；负值使原点向下，向左移动）；x_{bi} 和 y_{bi} 指明了在位图光栅化之后将 x 和 y 的增量加到光栅位置上（参见图 9-3）。

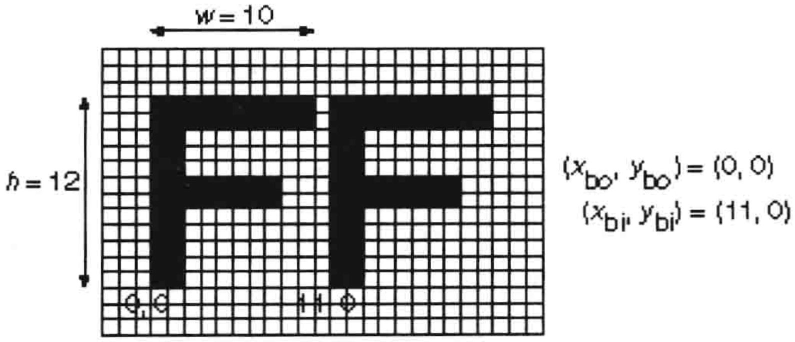


图 9-3　位图及其相关的参数

位图的原点可以进行任意设置，使得字符向原点下方扩展（常应用于下行字母，比如 g，j，和 y）和向原点的左方扩展（应用于各种旋转的字母，或者斜体字母）很容易便可以实现。

位图绘制后，当前光栅位置由 x_{bi} 和 y_{bi} 分别向 x 和 y 方向扩展。对于标准的拉丁字体，y_{bi} 取值 0.0 而 x_{bi} 取值为正（因为连续的字母都是由左向右画）。对于希伯来语，其字符是由右向左的，x_{bi} 的值就为负。对于按列垂直书写字符的字体，x_{bi} 应为 0，而 y_{bi} 不为 0。在图 9-3 中，每次绘出字符 F 当前的光栅位置移动 12 个像素点，在连续的字符间留有 2 个像素的空间。

既然 x_{bo}，y_{bo}，x_{bi} 和 y_{bi} 均为浮点数，字符也不必是像素宽度的整数倍。实际字符的绘制是严格按照像素边界来进行的，但当前光栅位置仍保持为浮点数，这样每个字符之间可以排列得尽量紧密。例如，将字符 F 例子中的 x_{bi} 由 12 改为 11.5，如果有更多的字符要绘制，那

么在字符之间的空间就变为 1 到 2 之间的数，可以大约估计为 1.5 个像素空间。但要记住，位图不能应用在可旋转的字体上，因为位图总是沿着帧缓冲器的 x 和 y 轴绘制的。

9.1.3 字体和显示列表

显示列表的相关内容我们在第 5 章已经做了详尽的讨论。其中一些显示列表命令与字符串的绘制有很特殊的关系。在阅读这部分的时候，请记住，对于使用位图和使用几何原语（包括点、线和多边形）绘制出的字符，这里所提出的方法应用后会有相同的效果。“执行多显示列表”中给出了一个几何原语字体的例子。

一种字体通常是由一个字符集合和一种绘制方法组成的，而且每一个字符都有一个标识码（通常是 ASCII 码）。对于标准的 ASCII 码表，大写字母 A 的标识数是 65，B 是 66，等等。字符串“DAB”就由三个索引值 68，65，66 代表。用最简单的近似可以这样来实现：显示列表数为 65 时将绘制 A，66 绘制 B，执行相应的显示列表就可画出字符串 68，65，66。这种方法也适用于命令 glCallLists()。

---

void glCallLists(GLsizei n, GLenum type, const GLvoid *lists);

此函数中，第一个参数 n 指明将要画的字符数；参数 type 通常是 GL_BYTE；lists 是一个字符码数组。

在很多应用当中，都需要用不同的字体和不同尺寸绘制字符，所以这种最简单的方法是不方便的。我们希望无论对于哪一种字体，65 均代表着 A。虽然可以强制将字体 1 的字符 A，B，C 编码为 1065，1066 和 1067，字体 2 的 A，B，C 编码为 2065，2066 和 2067，但超过 256 的数便无法在一个 8 位的字节中存储了。解决方案是，给在每一个字符串登录时，加一个偏移量并选择显示列表。在这种情况下，字体 1 的 A，B，C 由 1065，1066 和 1067 代表，在字体 2 中，相应的由 2065，2066 和 2067 代表。然后，用字体 1 画字符的时候，设置偏移量为 1000 并用显示列表画 65，66，67。当改用字体 2 的时候，可以使用同一显示列表，而只需将偏移量设置为 2000。

使用这种方法的时候，需要用命令 glListBase()来设置偏移量。对于前述例子，可以把 1000 或 2000 作为惟一的参数来调用。那么我们需要的就是相邻的未被占用的显示列表数，而这一点可以通过命令 glGenLists()得到。

GLuint glGenLists(GLsizei range);

该函数返回一组 range 个显示列表标识符。返回的列表无论是否为空，都标志为“used”，这样可以保证以后的调用中不会返回相同的列表标识（除非这一列表预先已经明确地删除）。若以 4 为参数调用 glGenLists()，得到的返回值为 81，那么就可以使用 81，82，83 和 84 作

为字符的显示列表标识符。如果 glGenLists()找不到要求长度的、未被使用的标识符，则返回值为 0。

注意：命令 glGenLists()将很容易一次性地删除与字体相关的列表。

大部分的欧美字体字符较少（少于 256），因此可存储在单字节中的编码种类足以代表每个字符。亚洲的字体需要更大的字符集，这样一个字节的编码就无法表达所有的字符。OpenGL 允许字符串由单字节、双字节、三字节甚至四字节的字符组成，这可以由命令 glCallLists()的类型参数指定。

该类型参数可以取如下的值：

```
GL_BYTE             GL_UNSIGNED_BYTE
GL_SHORT            GL_UNSIGNED_SHORT
GL_INT              GL_UNSIGNED_INT
GL_FLOAT            GL_2_BYTES
GL_3_BYTES          GL_4_BYTES
```

关于这些值的详细讨论请参见执行多显示列表。

9.1.4　一个完整的字体定义和使用

本节中的例程，使用上一节中描述的命令 glBitmap()和显示列表定义了一个完整的光栅字体。同时定义了完整的 ASCII 字体（尽管在本例中并无必要），这一字体中所有字符都为同一宽度。该例程代码与字符 F 的例子基本相同，只是在这里使用了 95 个位图（每个对应着包括空格在内的可打印的 ASCII 字符）并且每个字符在显示列表中对应着一条命令。每一个显示列表的标识符等于该字符的 ASCII 码，并且所有的字符都有相同的偏移量。

例 9-2 将产生如图 9-4 所示的输出。

图 9-4　定义一个完整的字体

例 9-2 定义一个完整的字体：font.c

```
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void makeRasterFont(void);
void printString(char *s);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLubyte rasters[][13] = {
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x18, 0x18, 0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x36, 0x36, 0x36, 0x36},
{0x00, 0x00, 0x00, 0x66, 0x66, 0xff, 0x66, 0x66, 0xff, 0x66, 0x66, 0x00, 0x00},
{0x00, 0x00, 0x18, 0x7e, 0xff, 0x1b, 0x1f, 0x7e, 0xf8, 0xd8, 0xff, 0x7e, 0x18},
{0x00, 0x00, 0x0e, 0x1b, 0xdb, 0x6e, 0x30, 0x18, 0x0c, 0x76, 0xdb, 0xd8, 0x70},
{0x00, 0x00, 0x7f, 0xc6, 0xcf, 0xd8, 0x70, 0x70, 0xd8, 0xcc, 0xcc, 0x6c, 0x38},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x1c, 0x0c, 0x0e},
{0x00, 0x00, 0x0c, 0x18, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x18, 0x0c},
{0x00, 0x00, 0x30, 0x18, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x18, 0x30},
{0x00, 0x00, 0x00, 0x00, 0x99, 0x5a, 0x3c, 0xff, 0x3c, 0x5a, 0x99, 0x00, 0x00},
{0x00, 0x00, 0x00, 0x18, 0x18, 0x18, 0xff, 0xff, 0x18, 0x18, 0x18, 0x00, 0x00},
{0x00, 0x00, 0x30, 0x18, 0x1c, 0x1c, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x60, 0x60, 0x30, 0x30, 0x18, 0x18, 0x0c, 0x0c, 0x06, 0x06, 0x03, 0x03},
{0x00, 0x00, 0x3c, 0x66, 0xc3, 0xe3, 0xf3, 0xdb, 0xcf, 0xc7, 0xc3, 0x66, 0x3c},
{0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x78, 0x38, 0x18},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x18, 0x0c, 0x06, 0x03, 0xe7, 0x7e},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0x07, 0x03, 0x03, 0xe7, 0x7e},
{0x00, 0x00, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0xff, 0xcc, 0x6c, 0x3c, 0x1c, 0x0c},
```

```
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0xfe, 0xc0, 0xc0, 0xc0, 0xc0, 0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
{0x00, 0x00, 0x30, 0x30, 0x30, 0x30, 0x18, 0x0c, 0x06, 0x03, 0x03, 0x03, 0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xe7, 0x7e, 0xe7, 0xc3, 0xc3, 0xe7, 0x7e},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x03, 0x7f, 0xe7, 0xc3, 0xc3, 0xe7, 0x7e},
{0x00, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x30, 0x18, 0x1c, 0x1c, 0x00, 0x00, 0x1c, 0x1c, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x06, 0x0c, 0x18, 0x30, 0x60, 0xc0, 0x60, 0x30, 0x18, 0x0c, 0x06},
{0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x60, 0x30, 0x18, 0x0c, 0x06, 0x03, 0x06, 0x0c, 0x18, 0x30, 0x60},
{0x00, 0x00, 0x18, 0x00, 0x00, 0x18, 0x18, 0x0c, 0x06, 0x03, 0xc3, 0xc3, 0x7e},
{0x00, 0x00, 0x3f, 0x60, 0xcf, 0xdb, 0xd3, 0xdd, 0xc3, 0x7e, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0x66, 0x3c, 0x18},
{0x00, 0x00, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
{0x00, 0x00, 0x7e, 0xe7, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
{0x00, 0x00, 0xfc, 0xce, 0xc7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc7, 0xce, 0xfc},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xc0, 0xff},
{0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xcf, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
{0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x7e},
{0x00, 0x00, 0x7c, 0xee, 0xc6, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06},
{0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xe0, 0xf0, 0xd8, 0xcc, 0xc6, 0xc3},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xdb, 0xff, 0xff, 0xe7, 0xc3},
{0x00, 0x00, 0xc7, 0xc7, 0xcf, 0xcf, 0xdf, 0xdb, 0xfb, 0xf3, 0xf3, 0xe3, 0xe3},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xe7, 0x7e},
{0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
{0x00, 0x00, 0x3f, 0x6e, 0xdf, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0x66, 0x3c},
{0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0xe0, 0xc0, 0xc0, 0xe7, 0x7e},
{0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
{0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
{0x00, 0x00, 0xc3, 0xe7, 0xff, 0xff, 0xdb, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
```

```
{0x00, 0x00, 0xc3, 0x66, 0x66, 0x3c, 0x3c, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3},
{0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x7e, 0x0c, 0x06, 0x03, 0x03, 0xff},
{0x00, 0x00, 0x3c, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x3c},
{0x00, 0x03, 0x03, 0x06, 0x06, 0x0c, 0x0c, 0x18, 0x18, 0x30, 0x30, 0x60, 0x60},
{0x00, 0x00, 0x3c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x3c},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc3, 0x66, 0x3c, 0x18},
{0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x38, 0x30, 0x70},
{0x00, 0x00, 0x7f, 0xc3, 0xc3, 0x7f, 0x03, 0xc3, 0x7e, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xfe, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0},
{0x00, 0x00, 0x7e, 0xc3, 0xc0, 0xc0, 0xc0, 0xc3, 0x7e, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x7f, 0xc3, 0xc3, 0xc3, 0xc3, 0x7f, 0x03, 0x03, 0x03, 0x03, 0x03},
{0x00, 0x00, 0x7f, 0xc0, 0xc0, 0xfe, 0xc3, 0xc3, 0x7e, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x30, 0x30, 0x30, 0x30, 0x30, 0xfc, 0x30, 0x30, 0x30, 0x33, 0x1e},
{0x7e, 0xc3, 0x03, 0x03, 0x7f, 0xc3, 0xc3, 0xc3, 0x7e, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0xc0, 0xc0, 0xc0, 0xc0},
{0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x00, 0x00, 0x18, 0x00},
{0x38, 0x6c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x00, 0x00, 0x0c, 0x00},
{0x00, 0x00, 0xc6, 0xcc, 0xf8, 0xf0, 0xd8, 0xcc, 0xc6, 0xc0, 0xc0, 0xc0, 0xc0},
{0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x78},
{0x00, 0x00, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xfe, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xfc, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x7c, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0x7c, 0x00, 0x00, 0x00, 0x00},
{0xc0, 0xc0, 0xc0, 0xfe, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0x00, 0x00, 0x00, 0x00},
{0x03, 0x03, 0x03, 0x7f, 0xc3, 0xc3, 0xc3, 0xc3, 0x7f, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xe0, 0xfe, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xfe, 0x03, 0x03, 0x7e, 0xc0, 0xc0, 0x7f, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x1c, 0x36, 0x30, 0x30, 0x30, 0x30, 0xfc, 0x30, 0x30, 0x30, 0x00},
{0x00, 0x00, 0x7e, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xc3, 0xe7, 0xff, 0xdb, 0xc3, 0xc3, 0xc3, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xc3, 0x66, 0x3c, 0x18, 0x3c, 0x66, 0xc3, 0x00, 0x00, 0x00, 0x00},
{0xc0, 0x60, 0x60, 0x30, 0x18, 0x3c, 0x66, 0x66, 0xc3, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xff, 0x60, 0x30, 0x18, 0x0c, 0x06, 0xff, 0x00, 0x00, 0x00, 0x00},
```

```
{0x00, 0x00, 0x0f, 0x18, 0x18, 0x18, 0x38, 0xf0, 0x38, 0x18, 0x18, 0x18, 0x0f},
{0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18},
{0x00, 0x00, 0xf0, 0x18, 0x18, 0x18, 0x1c, 0x0f, 0x1c, 0x18, 0x18, 0x18, 0xf0},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x06, 0x8f, 0xf1, 0x60, 0x00, 0x00, 0x00}
};

GLuint fontOffset;

void makeRasterFont(void)
{
    GLuint i;
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    fontOffset = glGenLists (128);
    for (i = 32; i < 127; i++) {
    glNewList(i+fontOffset, GL_COMPILE);
    glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0, rasters[i-32]);
    glEndList();
    }
}

void myinit(void)
{
    glShadeModel (GL_FLAT);
    makeRasterFont();
}

void printString(char *s)
{
    glPushAttrib (GL_LIST_BIT);
    glListBase(fontOffset);
    glCallLists(strlen(s), GL_UNSIGNED_BYTE, (GLubyte *) s);
    glPopAttrib ();
}
```

```
void CALLBACK display(void)
{
   GLfloat white[3] = { 1.0, 1.0, 1.0 };
   int i, j;
   char teststring[33];

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3fv(white);
   for (i = 32; i < 127; i += 32) {
   glRasterPos2i(20, 200 - 18*i/32);
   for (j = 0; j < 32; j++)
       teststring[j] = (char) (i+j);
   teststring[32] = 0;
   printString(teststring);
   }
   glRasterPos2i(20, 100);
   printString("Life is too short to");
   glRasterPos2i(20, 82);
   printString("register every wrongdoing.");
   glFlush ();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho (0.0, w, 0.0, h, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
   auxInitPosition (0, 0, 350, 350);
```

```
    auxInitWindow ("编程指南例 9-2");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

9.2　图像

图像同位图相似，只是不像位图那样在屏幕的矩形区内仅用一位来标识像素点，图像包含了更多的信息。例如，图像可以包含存储在每一像素点中完整的四元组(R, G, B, A)值。一幅图像可由如下途径得到：

- 经过扫描仪数字化的相片。
- 由图形程序通过图形硬件在屏幕上产生，然后再逐个像素点读回的图像。
- 由软件程序在内存中逐个像素点产生的图像。

通常所认为的图像同图片一样，来自颜色缓存。也可以把一个矩形区的像素数据写入和读出深度缓存或模板缓存。对这些缓存的解释请参见第 11 章。

除了简单地在屏幕上进行显示之外，图像可以用作纹理图，这样就可以将图像粘贴到多边形上，并且在屏幕上以正常的方式进行渲染。关于这项技术的详细讨论请参见第 10 章。

9.2.1　像素数据的读、写和拷贝

OpenGL 提供了处理图像数据的三个基本命令：

- glReadPixels()将像素的矩形阵列数据从帧缓存中读出并存储到处理器内存中。
- glDrawPixels()将保存在处理器内存中的像素数据写入帧缓存。
- glCopyPixels()将像素阵列数据从帧缓存的某一块拷贝到另一块。这条命令的功能同依次调用 glReadPixels()和 glDrawPixels()相近，但后者并不将数据写入处理器内存。

虽然这些命令的基本思路并不复杂，但是由于帧缓存数据的多样性，像素信息在计算机内存中存储方式的多样性，以及在读、写和拷贝操作的过程中数据转换方法的多样性，使得其应用中复杂性大大增加。所有的这些不同便可能转变为多种操作模式。如果应用程序的所有工作仅是拷贝屏幕上的图像，或暂时将其读进内存以准备将来能被拷贝出去，则大部分操作模式都可以忽略不计。但是，如果希望在内存中修改数据——例如，如果以某种格式存储了图像，而窗口需要的却是另一种格式——或如果想将图像数据存储到文件当中，以备下一次可以重新调用或存储到另一个图形特性显著不同的机器上，则必须深入理解各种操作模式。

本节详细讨论了这几个命令。像素的存储、变换和映射这一部分则详细讨论了像素的存

储模式、像素的变换操作和像素的映射操作。

---

void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format,GLenum type, GLvoid *pixels);

此函数从帧缓存矩形中读取像素数据，该矩形的左下角在点(x, y)，尺寸为 width 和 height，并存储在指针 pixels 所指向的数组中。参数 format 指明了要读取的像素数据的格式（一个索引值或 R, G, B、A 分量值，如表 9-1 所列），而 type 则指明了每一元素的数据类型(见表 9-2)。

表 9-1 在 glReadPixels() 或 glDrawPixels()函数中使用的像素格式

名 称	像素数据类型
GL_COLOR_INDEX	单色索引
GL_RGB	RGB 三色分量
GL_RGBA	RGBA 四色分量
GL_RED	红色分量
GL_GREEN	绿色分量
GL_BLUE	蓝色分量
GL_ALPHA	Alpha 分量
GL_LUMINANCE	亮度分量
GL_LUMINANCE_ALPHA	亮度分量和 alpha 分量
GL_STENCIL_INDEX	模板索引
GL_DEPTH_COMPONENT	深度分量

表 9-2 glReadPixels() 和 glDrawPixels()的数据类型

名称	数据类型
GL_UNSIGNED_BYTE	无符号的 8 位整型值
GL_BYTE	有符号的 8 位整型值
GL_BITMAP	无符号 8 位整型值
GL_UNSIGNED_SHORT	无符号的 16 位整型值
GL_SHORT	有符号的 16 位整型值
GL_UNSIGNED_INT	无符号的 32 位整型值
GL_INT	32 位整型值
GL_FLOAT	单精度浮点数

void glDrawPixels(GLsizei width, GLsizei height, GLenum format, GLenum type, const GLvoid *pixels);

该函数绘制一个尺寸为 width × height 的像素数据矩形。像素矩形的左下角位于当前光栅位置。参数 format 和 type 与函数 glReadPixels()具有相同的意义。由指针 pixels 所指向的数组中包含了所要绘制的像素数据。如果当前的光栅位置是无效的，则进行的是空操作，并在调用之后仍保持无效。

这里需要记住一点，每次读写一个到四个之间几个要素完全依赖于数据的格式。例如，如果格式为 GL_RGBA，而读进的是 32 位的整型值（即 type 为 GL_INT），那么每读取一个像素需要 16 字节的存储空间（4 分量×4 字节/分量）。

所存储图像的每一要素在内存中的存储方式如表 9-2 所示。如果一个要素代表一个连续值，比如红，绿，蓝或亮度分量，则每个值要按照可用的位数来进行缩放。例如，红色分量是一个值在 0.0 到 1.0 之间的浮点数。如果需要压缩为一个无符号字节，即使在帧缓存中为这个红色分量分配了更多的位数，也仅保留 8 位的精度。GL_UNSIGNED_SHORT 和 GL_UNSIGNED_INT 分别给出 16 位和 32 位的精度。而 GL_BYTE, GL_SHORT 和 GL_INT 的有符号数分别具有 7，15 和 31 位的精度，缺少的一位是因为负值通常不用。

如果要素是一个索引值（比如，颜色索引或模板索引），而其类型并不是 GL_FLOAT，其值仅简单地用该类型中的可用位来屏蔽。GL_BYTE, GL_SHORT 和 GL_INT 这类有符号的形式，则少屏蔽一位。例如，如果一个颜色索引被存储在一个有符号的 8 位整型数中，它先被 0xff 屏蔽，掩码包括 7 个 1。如果类型为 GL_FLOAT，该索引则被简单地换算成一个单精度的浮点数（比如，索引数 17 只简单地变成 17.0）。

void glCopyPixels(GLint x, GLint y,　GLsizei width, GLsizei height, GLenum type);

此函数从帧缓存矩形区域中拷贝像素数据，该帧缓存矩形的左下角为(x, y)，尺寸为 width 和 height。数据被拷贝到一个新的矩形区域中，其左下角坐标由当前光栅位置指定。参数 type 可以是 GL_COLOR, GL_STENCIL 或 GL_DEPTH。当 type 转换为以下形式之一来对参数进行格式化时，glCopyPixels()与连续调用 glReadPixels()和 glDrawPixels()功能相近。

- 如果 type 为 GL_DEPTH 或 GL_STENCIL，那么就要分别使用 GL_DEPTH_COMPONENT 或 GL_STENCIL_INDEX;
- 如果指定了 GL_COLOR，则根据系统是 RGBA 模式还是颜色索引模式分别使用 GL_RGBA 或 GL_COLOR_INDEX。

在 glReadPixels()作用期间，glCopyPixels()可以应用于各种像素变换，转换函数等。得到的结果数据的存储类似 glDrawPixels()，但并不应用第二次变换。需要注意的是，对于

glCopyPixels()，因为数据并不拷贝到处理器的内存中，所以并不需要格式或数据参数。对于这三个函数，在对帧缓存进行数据读写时的确切转换，取决于当时正作用的模式。关于这一问题细节的讨论可参见下一节。

9.2.2 图像的放大与缩小

通常，图像中的每个像素对应写入屏幕上的一个像素。但是，OpenGL 也提供了命令 glPixelZoom()，用以对图像进行任意的放大和缩小。

---

void glPixelZoom(GLfloat zoomx, GLfloat zoomy);

该函数在 x 和 y 方向上设置每一次像素写操作的放大和缩小因子。zoomx 和 zoomy 的缺省值均为 1.0。如果是 2.0，每个图像像素就将被画到 4 个屏幕像素上去。注意，放大和缩小因子还可以是小数或负数。

在光栅化的过程当中，每一个图像像素均被视为一个 zoomx×zoomy 的四边形，所有中心位于四边形之内的像素点会产生片原。为了明确说明，将（xrp，yrp）设置为当前光栅位置，当一组特殊的要素（索引或分量）位于第 n 行第 m 列时，考虑在窗口坐标系中的矩形区域，其中心位于：

(xrp + zoomx n, yrp + zoomy m) 和 (xrp + zoomx(n+1), yrp + zoomy(m+1))

位于该矩形区内（或在其底边和左边上）的任何片原都产生于对这一组特定要素的响应。

9.3 像素的存储、变换和映射

本节将详细讨论关于像素的存储和转换模式，包括如何设置任意的映射来实现变换过程中的转换。需要注意的是，只有在将像素数据从一种格式转换到另一种格式的时候才需要使用这些模式。

9.3.1 像素模式概论

存储在内存当中的图像为在屏幕矩形区中的每一个像素包含 1 到 4 个数据块（称为要素）。数据可以仅包含颜色索引或亮度（亮度是红、绿、蓝三色的加权和），或者对每一像素点包含红、绿、蓝以及 alpha 分量。对像素数据的排列，即格式，决定了为每一像素存储的要素数目及其顺序。

有些要素（比如，颜色索引或模板索引）是整型量，而其他的要素（比如，红、绿、蓝以及 alpha 分量或者深度分量）是 0.0 到 1.0 之间的浮点值。浮点分量通常存储为位平面的固定点，且常比完全浮点数所需的分辨率要低（比如，对于颜色分量用 8 位）。用以表示各分量的确切位数取决于使用的特殊硬件。因此，把每一分量都存储为一个全 32 位的浮点数

就显得十分浪费，尤其要考虑通常的图像都包括上百万个像素点。

要素可以在内存中以各种数据类型存储，从 8 位的整型数到 32 位的整型数甚至浮点数。OpenGL 明确定义了每一分量从一种格式到另一种数据类型的转换。但要注意，将一个高分辨率分量存储为一个较少位数的类型时，可能丢失数据。

图像数据通常以二维或三维数组的形式存储在处理器内存中。我们常常希望能够显示或存储相应于一个子矩形数组的子图像。而且需要考虑，不同的计算机具有不同的字节次序约定。最后，如果数据在处理器内存中以 2 字节，4 字节或 8 字节等固定方式排列，一些计算机的硬件将会使帧缓存对数据的读写更为便捷。对于这类计算机，需要控制字节的排列。这里提出的问题全部是由像素存储模式来控制的；同前面的两个例程一样，可以使用 glPixelStore*()命令来指定这些模式。

OpenGL 提供了几种将图像从内存移到帧缓存，或从帧缓存移到内存的操作。例如，可以改变分量的范围——通常情况下，红色分量在 0.0 到 1.0 之间，程序员可以根据自己的需要改变其区间，或者根据来自不同图形系统的数据将红色的分量存储为 0～1 以外的区间。甚至可以创建一系列图，在像素的传递过程中任意转换颜色索引和颜色分量。这些转换在像素传递出入帧缓存的过程中进行，称作像素传递模式。该模式由 glPixelTransfer*() 和 glPixelMap*()命令来控制。

其他可以进行控制的模式还包括读取像素的帧缓存，在写入帧缓存时对像素进行的放大等等。

最后要清楚的一点是，尽管颜色、深度和模板缓存有许多相似之处，但并不完全相同，而且一些模式对特殊的缓存有特殊的应用。所有这些模式的细节，包括那些特殊的应用，将在接下来的几节中讨论。

9.3.2　像素存储控制模式

所有的像素存储模式都由命令 glPixelStore*()来进行控制。通常可以通过该命令的连续调用来设置参数的值。

---

void glPixelStore{if}(GLenum pname, TYPEparam);

该命令用于设置像素存储模式，它将影响 glDrawPixels*()，glReadPixels*()，glBitmap()，glPolygonStipple()，glTexImage1D()，glTexImage2D() 和 glGetTexImage()的操作。pname 可能用到的参数名称见表 9-3，表中其余的各项分别是数据的类型，初始值和取值区间。参数 GL_UNPACK*控制着数据是如何通过 glDrawPixels*()，glBitmap()，glPolygonStipple()，glTexImage1D() 和 glTexImage2D()命令从内存中解压的。参数 GL_PACK*则控制数据是如何由 glReadPixels*() 和 glGetTexImage()命令压缩到内存的。

表 9-3 Pname 参数说明

参数名称	类型	初值	有效范围
GL_UNPACK_SWAP_BYTES, GL_PACK_SWAP_BYTES	GLboolean	FALSE	TRUE/FALSE
GL_UNPACK_LSB_FIRST, GL_PACK_LSB_FIRST	GLboolean	FALSE	TRUE/FALSE
GL_UNPACK_ROW_LENGTH, GL_PACK_ROW_LENGTH	GLint	0	任意非负整型数
GL_UNPACK_SKIP_ROWS, GL_PACK_SKIP_ROWS	GLint	0	任意非负整型数
GL_UNPACK_SKIP_PIXELS, GL_PACK_SKIP_PIXELS	GLint	0	任意非负整型数
GL_UNPACK_ALIGNMENT, GL_PACK_ALIGNMENT	GLint	4	1, 2, 4, 8

由于相应于压缩和解压的参数有相同的意义，在本节余下部分将对这两个操作一起进行讨论，而省略各参数的 GL_PACK 或 GL_UNPACK 前缀。比如，*SWAP_BYTES 是统指 GL_PACK_SWAP_BYTES 和 GL_UNPACK_SWAP_BYTES。

如果参数*SWAP_BYTES 的值为 FALSE（缺省情况下），内存中字节的次序是 OpenGL 客户原来的次序；否则，字节逆序。逆序字节适用于任何大小的要素。

注意：*如果 OpenGL 应用程序不与其他计算机共享图像，可以忽略字节顺序的问题。如果读者的应用程序必须渲染一个由其他计算机创建的图像，而两机的某些机制不同，字节的顺序就可以用*SWAP_BYTES 来交换。*

参数*LSB_FIRST 适用于绘制或读取 1 位的图像或位图，因为这些图像和位图都是用 1 位来进行每个像素的存储和恢复的。如果*LSB_FIRST 的值为 FALSE（缺省情况下），在字节中从最重要的位开始读按从左到右的顺序读取各二进制位；反之，就采用相反的顺序。例如，如果*LSB_FIRST 的值为 FALSE，而相关的字节是 0x31，各位的顺序是：{0, 0, 1, 1, 0, 0, 0, 1}。如果*LSB_FIRST 的值为 TRUE，则各位的顺序为：{1, 0, 0, 0, 1, 1, 0, 0}。

有时我们希望画或读取存储在内存中的整个图像数据矩形区的一个子矩形区。如果内存中的矩形区大于要画出或读取的区域，就需要用*ROW_LENGTH 来指定较大矩形的实际长度。如果*ROW_LENGTH 为 0（缺省值），行的长度就认为是与 glReadPixels*()，glDrawPixels*(), 或 glCopyPixels()所指定的宽度相同。

在开始拷贝子矩形区数据之前，还需要指定要跳跃（skip）的行和像素的数目。这些数目是由*SKIP_ROWS 和 *SKIP_PIXELS 指定的，如图 9-5 所示。在缺省情况下，两个参数的值均为 0，也就是从整个区域的左下角开始。

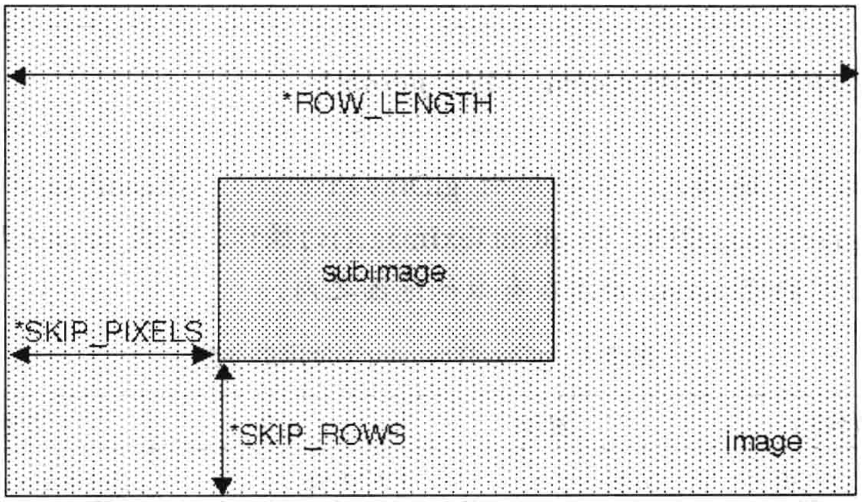


图 9-5　参数*SKIP_ROWS, *SKIP_PIXELS 和 *ROW_LENGTH

通常，如果存储在内存中的数据按某种专门的字节顺序排列，将优化计算机的特定硬件从内存中对像素数据的读取和写入。比如，在 32 位字节的机器中，如果初始数据以 32 位为界顺序排列，也就是通常具有 4 的倍数的地址，硬件能够更快地获取数据。同样，如果以 8 字节为界来排列数据，64 位结构会以更快的速度工作。而对有些计算机，字节如何排列并没有什么区别。

举个例子，假设计算机在像素数据以 4 位为边界排列时工作状况良好。如果强迫图像的每一列数据在 4 位边界处开始存储，图像存储的效率将大大提高。如果图像是 5 个像素的宽度，而每一像素包括各一个字节的红、绿、蓝颜色信息，那么一行就需要存储 5×3=15 个字节的数据。如果从第一行开始每一行都以 4 字节为边界存储数据，可以达到最大的显示效率，因此内存对每一行的存储都浪费了一个字节。如果数据这样进行存储，需要合理设置参数*ALIGNMENT（在这种情况下设置为 4）。

如果*ALIGNMENT 设置为 1，就会用到相邻的可用字节。如果设置为 2，为了使下一行的第一个字节地址是 2 的倍数，在每一行的结尾需要忽略掉一个字节。对于位图的情况，每一个像素点只存储一位，尽管每一位都必须计入，同样的字节排列依然有效。比如，如果为每一像素点存储一位，行的长度是 75，而排列为 4，每一行需要 75/8，或 93/8 个字节。由于 12 是大于 93/8 最小的 4 的倍数，因此每一行都要占用 12 字节的内存。

9.3.3　像素传递操作

在像素传递出入帧缓存时，可以对像素做各种操作。那些连续的分量，包括红、绿、蓝、alpha 和深度分量，可以应用映射变换。而且，在变换之后，这些分量同颜色索引、模板值等一样，均可通过对表的任意查找进行变换。

一些像素传递函数的特征值可由 glPixelTransfer*()来设置。其他的特征值由 glPixelMap*()指定，该函数将在下一部分讨论。

void glPixelTransfer{if}(GLenum pname, TYPE param);

该函数设置像素传递模式，它将影响 glDrawPixels*(), glReadPixels*(), glCopyPixels(), glTexImage1D(), glTexImage2D() 和 glGetTexImage()的操作。参数 pname 取值是表 9-4 第一列所给出的，并且它的值 param 必须在有效的范围之内。

表 9-4 glPixelTransfer*()使用的参数

参数名称	类型	初值	有效范围
GL_MAP_COLOR	GLboolean	FALSE	TRUE/FALSE
GL_MAP_STENCIL	GLboolean	FALSE	TRUE/FALSE
GL_INDEX_SHIFT	GLint	0	(-∞ ， +∞)
GL_INDEX_OFFSET	GLint	0	(-∞ ， +∞)
GL_RED_SCALE	GLfloat	1.0	(-∞ ， +∞)
GL_GREEN_SCALE	GLfloat	1.0	(-∞ ， +∞)
GL_BLUE_SCALE	Glfloat	1.0	(-∞ ， +∞)
GL_ALPHA_SCALE	GLfloat	1.0	(-∞ ， +∞)
GL_DEPTH_SCALE	GLfloat	1.0	(-∞ ， +∞)
GL_RED_BIAS	GLfloat	0	(-∞ ， +∞)
GL_GREEN_BIAS	GLfloat	0	(-∞ ， +∞)
GL_BLUE_BIAS	GLfloat	0	(-∞ ， +∞)
GL_ALPHA_BIAS	GLfloat	0	(-∞ ， +∞)
GL_DEPTH_BIAS	GLfloat	0	(-∞ ， +∞)

如果参数 GL_MAP_COLOR 或 GL_MAP_STENCIL 为 TRUE，则映射被激活。关于如何进行映射以及如何改变图的内容，请参见下一节的内容。其他参数将直接影响像素分量的值。

像素从帧缓存数据读到内存中时执行的像素转换，与其反过程——内存到帧缓存（画的过程）基本相似但并不完全相同，下面将解释这一点。

9.3.4 像素矩形

图 9-6 和下面的段落描述了将像素绘制到帧缓存中的操作过程。

1）如果像素不是索引值（颜色或模板），在必要情况下，第一步要将各分量转换为浮点数。详细的情况请参见表 9-1。

2）如果格式为 GL_LUMINANCE 或 GL_LUMINANCE_ALPHA，通过对 R，G，B 每个成分应用亮度要素，将该要素转换为所对应的 R，G，B 分量值。在 GL_LUMINANCE_ALPHA 格式下，alpha 值变为 A 值。如果没有 A 值，将 alpha 设置为 1.0。

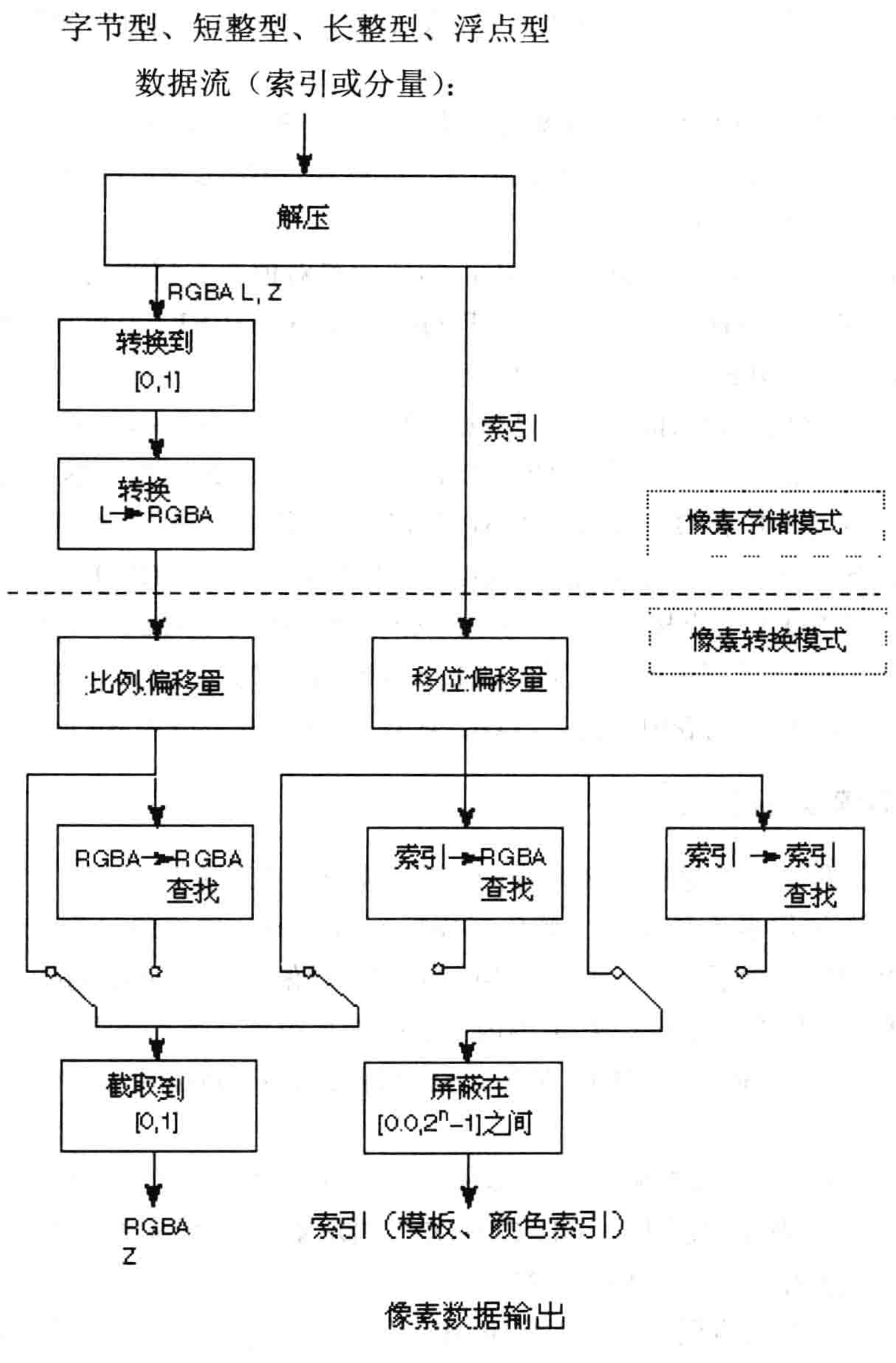


图 9-6　用 glDrawPixels*()绘制像素

3）将每一个分量（R，G，B，A 或 depth）乘以适当的缩放比例，并加上合适的偏移量。例如，将 R 分量乘以相应于 GL_RED_SCALE 的值，并加上相应于 GL_RED_BIAS 的值。

4）如果 GL_MAP_COLOR 为 TRUE，则将每个 R，G，B 和 A 分量截取在[0.0,1.0]的范围内，乘以一个比表的尺寸小 1 的整型值，截去顶端值，然后在表中查找对应。详细的论述请参见“像素映射”。

5）下一步，如果 R，G，B 和 A 分量值不在[0.0,1.0]范围内，则对其进行截取，并将其

转化为固定小数点（定点，相对于浮点），其数目等于在相应的帧缓存分量中二进制点左侧的位数。

6）如果使用索引值（模板索引或颜色索引），将这些值首先转化为二进制点右侧有一些未指定位数的定点数（如果它们的初始化值为浮点数）。初始化为定点数的索引仍保持不变，并将所有二进制点右侧的位置为 0。

7）结果的索引值由 GL_INDEX_SHIFT 的位的绝对值向右移或向左移；左移还是右移取决于参数 GL_INDEX_SHIFT 的正负——若 GL_INDEX_SHIFT > 0 则向左移，否则右移。最后，将 GL_INDEX_OFFSET 加到索引值上。

8）下面的一步将用到索引值，是否进行根据程序员采用的是 RGBA 模式还是索引值模式。在 RGBA 模式下，颜色索引值将用由 GL_PIXEL_MAP_I_TO_R，GL_PIXEL_MAP_I_TO_G，GL_PIXEL_MAP_I_TO_B 和 GL_PIXEL_MAP_I_TO_A 所指定的颜色分量转换为 RGBA 值（详细论述请参见下一节）。否则，如果 GL_MAP_COLOR 为 TRUE，颜色索引将在表 GL_PIXEL_MAP_I_TO_I 中查找。（如果 GL_MAP_COLOR 为 FALSE，索引值不改变）。如果图像是由模板索引而非颜色索引组成，则当 GL_MAP_STENCIL 为 TRUE 时，索引值在相应于 GL_PIXEL_MAP_S_TO_S 的表中查找；当 GL_MAP_STENCIL 为 FALSE 时，模板索引不变。

9.3.5 像素读取过程详解

在像素读取过程中，将进行许多同样的转换，如图 9-7 所示。

如果将要读取的像素不是索引（颜色或模板）值，各分量将映射到[0.0,1.0]范围内，即与写的过程恰好相反。然后，对各分量应用缩放比例和偏移量。如果 GL_MAP_COLOR 为 TRUE，将映射并再次截取到[0.0,1.0]范围内。如果期望得到亮度而不是 RGB，则将 R，G，B 各分量加在一起（L = R + G + B）。最后，结果将根据由 glPixelStore*()设置的 GL_PACK* 模式压入内存。

如果像素是索引（颜色或模板），将对其进行偏移，而且如果 GL_MAP_COLOR 为 TRUE，还应该应用映射。如果存储模式为 GL_COLOR_INDEX 或 GL_STENCIL_INDEX，像素索引将被屏蔽为存储类型（1，8，16 或 32）的位数并存入内存。如果存储格式为分量的一种（如亮度或 RGB），索引将根据“索引到 RGBA”映射表进行映射。然后，将按着其最初即是 RGBA 像素（包括可能转换成亮度），来进行处理。

缩放比例，偏差，移位和偏移值与那些绘制像素中使用的值相同，这样如果对像素既有读又有画的操作，要保证在做读或画操作之前将这些分量重新设置为适当的值。类似地，如果打算利用映射来进行读和画的操作，各种映射表（参见下一节）也必须正确地重设。

注意： 在读和画的操作中对亮度的处理显得有些别扭（其实并没有什么问题）。例如，亮度并不像所见到的那样总是等量依赖于 R，G，B 分量的值。如果希望亮度中 R 占 30%，G 占 59%而 B 占 11%，则应该将 GL_RED_SCALE 设置为 0.30，GL_RED_BIAS 设置为 0.0 等等。那么 L 的计算值就是 0.30R+0.59G+0.11B。

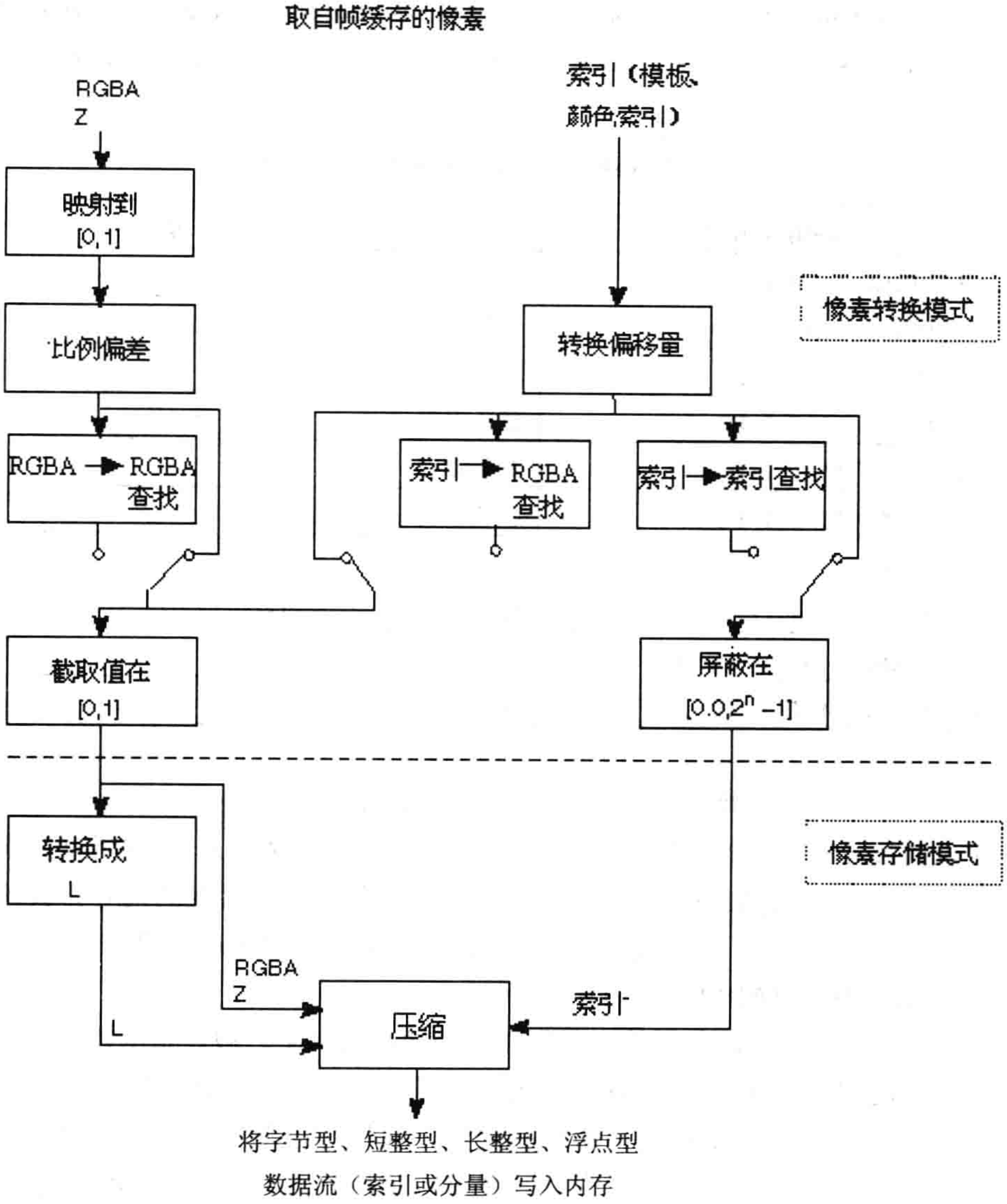


图 9-7　用 glReadPixels*()读取像素

9.3.6　像素映射

如前面一节中提到的那样，所有的颜色分量、颜色索引和模板索引都可以在放入屏幕内存之前通过表进行查询。控制该映射的命令是 glPixelMap*()。

---

void glPixelMap{ui us f}v(GLenum map, GLint mapsize, const TYPE *values);

该函数加载由 mapsize 个 map 确定的像素映射，其值由指针 values 指定；缺省的 mapsize 为 1 而缺省的 values 为 0。表 9-5 列出了映射的名称和值。每个映射的 mapsize 必须是 2 的幂。

表 9-5 glPixelMap*()所使用的有效参数名称和值

映射表名称	地址	值
GL_PIXEL_MAP_I_TO_I	颜色索引	颜色索引
GL_PIXEL_MAP_S_TO_S	模板索引	模板索引
GL_PIXEL_MAP_I_TO_R	颜色索引	R
GL_PIXEL_MAP_I_TO_G	颜色索引	G
GL_PIXEL_MAP_I_TO_B	颜色索引	B
GL_PIXEL_MAP_I_TO_A	颜色索引	A
GL_PIXEL_MAP_R_TO_R	R	R
GL_PIXEL_MAP_G_TO_G	G	G
GL_PIXEL_MAP_B_TO_B	B	B
GL_PIXEL_MAP_A_TO_A	A	A

映射表的最大尺寸是依赖于计算机的。可以使用 glGet*()命令来得到用户计算机所支持的像素映射表的尺寸：用查询参数 GL_MAX_PIXEL_MAP_TABLE 得到所有像素映射表的最大尺寸；用 GL_PIXEL_MAP_*_TO_*_SIZE 得到当前指定映射表的尺寸。其地址为颜色索引的 6 个映射表的尺寸必须被变换为 2 的幂。4 个 RGBA 映射表可以取从 1 到 GL_MAX_PIXEL_MAP_TABLE 之间的任何尺寸。

为了说明映射表是如何起作用的，我们看一个简单的例子。假设想创建一个有 256 个条目的表，使用 GL_PIXEL_MAP_I_TO_I 将颜色索引映射为颜色索引。首先为 0 到 255 之间的值各创建一个条目，并用 glPixelMap*()初始化。我们考虑使用该表的极限情况，例如，将 100 到 0 之间的所有索引映射为 0，将 101 到 255 的所有索引映射为 255。在这种情况下，表中包含 101 个 0 和 155 个 255。将 glPixelTransfer*()的参数 GL_MAP_COLOR 设置为 TRUE，激活像素映射表。一旦像素映射表载入并激活，输入的小于 101 的颜色索引映射为 0；在 101 到 255 之间的映射为 255。如果输入的像素大于 255，首先被 255 屏蔽，去掉所有 8 位以后的位，在表中查找屏蔽以后的值。如果输入的索引是浮点数如 88.14585，取整为最接近的整型值 88，然后在表中查找，为 0。

第 10 章　纹理映射

本章主题

到现在为止，本书中所涉及到的每个几何图元都是以单色、或者是在各个顶点颜色之间以平滑阴影来绘制的——也就是说，绘图时没有应用纹理映射。如果绘制一垛大的砖墙，不使用纹理映射，只好将每一块砖都画成一个独立的多边形。这样一来，一垛巨大的砖墙将需要成千上万块砖。即便如此，砖块也会由于太光滑和规则，显得很不真实。

把整个墙壁画成一个多边形，纹理映射能够把一幅砖墙的图像（可以通过对一张真实的墙壁像片进行扫描得到）粘贴到多边形上，并可以确保该矩形在变换和渲染时都是正确的：例如在透视图中观察该墙壁，墙砖的尺寸大小便会随着其与观察点距离的增加越来越小。纹理映射的其他应用还包括：把植被通过纹理映射贴到巨大的多边形上，以仿真飞行模拟器的地面；墙纸图案；利用纹理把多边形做成大理石、木材或织物等自然材质。纹理映射的应用范围是十分广泛的。将纹理粘贴到多边形上只是常见的形式，而事实上，纹理可以应用于所有的图元——包括点、线、多边形、位图和图像等。彩图 6、8、18~21，24~27 和 29~30 都演示了纹理的用法。

因此，纹理映射是一个涉及面广而又相当复杂的题目，而且，在使用纹理映射时又有多种编程方案。例如，可以把纹理映射到由一组多边形近似的表面或是弯曲的表面上，也可以在一个方向或两个方向上重复使用一种纹理来覆盖表面。纹理甚至可以是一维的。此外，还可以自动将纹理映射到物体上，利用纹理来表示所观察到的物体的轮廓或其他一些属性。可以给发亮的物体粘上纹理，使其看起来就好像是处在房间或其他环境的正中央，其表面反射的是周围环境的景色。最后，纹理还可以用不同的方式粘贴到物体表面上。既可以直接画（类似于往物体表面贴花），也可以用于调整物体表面的颜色，或者将纹理颜色和物体表面颜色进行混合。读者如果是首次接触纹理映射技术，可能会觉得本章的讨论速度稍嫌快了一点。如果是这样的话，可以参阅相关纹理映射参考书籍。

纹理是数据的简单矩阵排列。这些数据包括颜色数据、亮度（流明）数据或者颜色和 alpha 数据。纹理数组中的每一个独立的数值通常称为一个纹理单元（texel）。矩形纹理还可以粘贴到非矩形区域上，这使得纹理映射的应用显得更加灵活，而这种方式也是非常合理的。

图 10-1 说明了纹理映射的过程。图的左侧代表整个纹理，黑色的边框代表四边形的形状，其顶点将要粘贴到纹理的斑点上。当在屏幕上显示这个四边形时，可以通过各式各样的变换（如旋转、平移、缩放和投影）使四边形发生扭曲。图的右侧显示了贴上纹理的四边形经过变换后是如何显示在屏幕上的（注意，该四边形是凹的，OpenGL 不能对其进行正确的

渲染）。

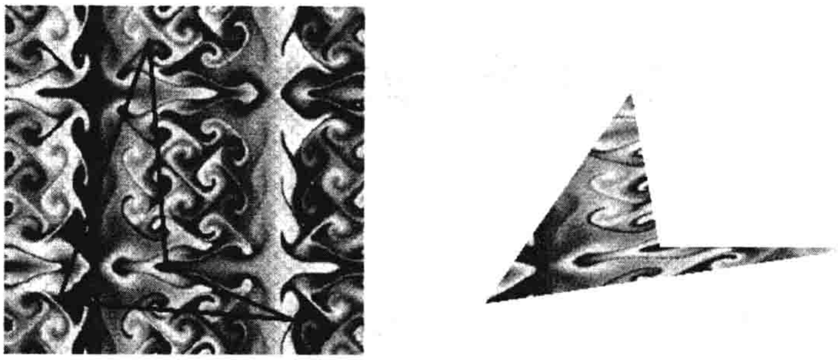


图 10-1 纹理映射过程

请注意纹理是如何发生扭曲来匹配四边形的扭曲的。在这种情况下，纹理在 x 方向上被拉长，而在 y 方向上被压缩，同时还发生了一定的旋转和剪切。根据纹理的大小、四边形的扭曲程度以及屏幕图像的大小，一些纹理单元可能映射到多个片原上，而一些片原可能被多个纹理单元所覆盖。由于纹理由离散的纹理单元组成（本例为 256×256 个单元），因此，将纹理单元映射到片原时必须进行过滤处理。例如，当多个纹理单元与一个片原相对应时，将对这些纹理单元取一个适当的平均；当纹理单元的边界落在片原的边界上时，就要对用到的纹理单元做加权平均处理。由于需要上述计算，因此纹理映射的计算代价是极其昂贵的，这也是为什么许多专用的图形系统必须包括支持纹理映射的硬件的原因。

本章从下面几个主要方面讲述了 OpenGL 的纹理映射的相关内容。

- “概述及范例”简要地给出了执行纹理映射的主要步骤，同时给出了一个相当简单的应用纹理映射的范例。
- “指定纹理”解释了如何指定二维或一维的纹理，同时还讨论了如何使用纹理边界，如何提供一系列不同尺寸的相关纹理，以及如何控制过滤方法，该方法用于确定如何将所用的纹理映射到屏幕坐标上。
- “调节和混合”讨论了在物体表面上绘制纹理的方法。当纹理映射不起作用时，可以用纹理颜色来代替纹理要用到的物体颜色，或者将这两种颜色混合成一种颜色。
- “分配纹理坐标”描述了如何计算纹理坐标，以及如何给物体顶点分配适当的纹理坐标。同时还解释了如何控制位于默认区域以外的坐标行为——也就是说，如何在表面上重复及截取纹理。
- “自动纹理坐标生成”向读者展示了如何利用 OpenGL 自动生成纹理坐标，以便实现诸如轮廓图案及环境图案类似的效果。
- “纹理的高级特性”解释如何操作纹理矩阵堆栈，以及如何使用纹理坐标 q。

记住，可以使用显示列表来存储所定义的纹理，以获得最大的运行效率。参见显示列表设计准则一节。

10.1　概述及范例

本节给出执行纹理映射所必需的步骤，同时给出了一个相当简单的纹理映射程序——所谓的简单，是因为纹理映射实际上是一个相当棘手的处理过程。

10.1.1　纹理映射的步骤

在使用纹理映射时，需要执行以下步骤：

1）指定纹理。

2）说明纹理如何应用于每一个像素点。

3）激活纹理映射。

4）提供纹理坐标及几何坐标来绘制场景。

注意：纹理映射只适用于 RGB 模式；在颜色索引模式下使用纹理映射，其结果是不可预料的。

1. 指定纹理

在最简单的情况下，纹理是一幅图像。正如大多数图像那样，纹理通常被认为是二维的，但是，纹理也可以是一维的。描述纹理的数据，每个纹理单元可以包括 1 到 4 个元素，用来表示从用一个常数，到用（R，G，B，A）四元组所能表示的任何事物。

使用一种称为 mipmapping 的高级技术，程序员可以用不同的分辨率来指定一个单一的纹理；比如说，利用这一特性可以避免在一堵远离观察点的墙上映射一幅满分辨率的砖墙图像，因为此时墙壁在屏幕上看起来就是一个像素点。而且，图像的定义可以包括边界值，以防止物体纹理坐标超出有效区域。边界值允许程序员把多个纹理映射图平滑地粘贴在一起，从而可以增加最大可用纹理的有效尺寸。

2. 对纹理如何应用于每一个像素点进行说明

程序员可以用下面的三种功能中的一种，根据片原和纹理图像数据中的颜色来计算最终的 RGBA 值：可以简单地用纹理颜色作为最终的颜色，这种方式如贴花一样，纹理被粘贴在片原的表面上；另外，可以用纹理来调整片原的颜色或颜色比例，该技术对光照和纹理的综合效果是非常有用的。还可以用纹理值将一个固定的颜色与片原的颜色融合在一起。

3. 激活纹理映射

在绘制场景之前需要激活纹理映射。激活或取消纹理映射的函数是 glEnable() 或 glDisable()，其参数可以为 GL_TEXTURE_1D 或 GL_TEXTURE_2D，分别代表一维和二维的纹理图。

4. 利用纹理坐标和几何坐标绘制场景

在粘贴纹理之前，必须说明纹理相对于片原是如何排列的。也就是说，必须指定场景中物体的纹理坐标和几何坐标。例如，对于二维纹理图，两个方向上纹理坐标均为[0.0，1.0]，

而要粘贴纹理的物体坐标却可以是任意的。在砖墙这个例子中，如果墙壁是正方形，并只想粘贴一个纹理，则该代码就有可能把纹理坐标（0, 0），（1, 0），（1, 1）和（0, 1）指定为墙壁的四个角。如果该墙壁很大，则需要在上面粘贴多个纹理。如果要这样做，就必须将纹理设计好，使得墙壁左侧的砖与右侧的砖正好融合到一起，同时使得上下的砖都相似。

同时还要指出如何处理[0.0，1.0]范围之外的纹理坐标。

10.1.2 一个简单的程序

一般情况下，用程序来生成纹理需要成百行代码，因此纹理通常从图像文件中读出。在例 10-1 中，纹理（由黑白相间的方块组成，类似于西洋跳棋盘）是由程序生成的。该程序将纹理用到两个正方形上，然后在透视图中进行渲染。其中一个正方形正对着观察点，而另一个向后倾斜 45° 角，如图 10-2 所示。在物体坐标中，两个正方形的大小是相同的。

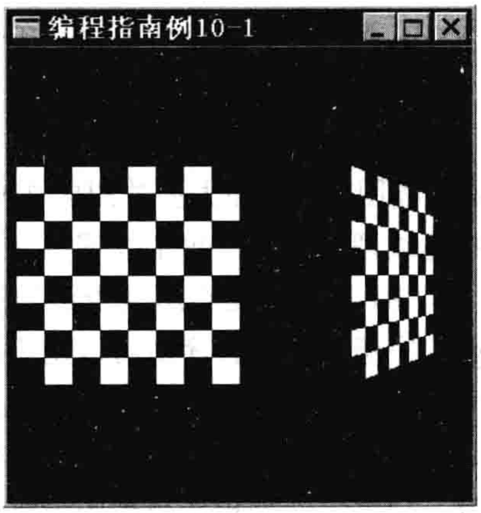


图 10-2 贴上纹理的正方形

例 10-1 纹理的简单使用：checker.c

```
#include "windows.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

#define    checkImageWidth 64
#define    checkImageHeight 64
GLubyte    checkImage[checkImageWidth][checkImageHeight][3];

void CALLBACK makeCheckImage(void)
{
```

```
    int i, j, r, c;
    for (i = 0; i < checkImageWidth; i++)
    {
        for (j = 0; j < checkImageHeight; j++)
        {
            c = ((((i&0x8)==0)^((j&0x8))==0))*255;
            checkImage[i][j][0] = (GLubyte) c;
            checkImage[i][j][1] = (GLubyte) c;
            checkImage[i][j][2] = (GLubyte) c;
        }
    }
}

void CALLBACK myinit(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    makeCheckImage();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexImage2D(GL_TEXTURE_2D, 0, 3, checkImageWidth,
                 checkImageHeight, 0,
                 GL_RGB, GL_UNSIGNED_BYTE,
                 &checkImage[0][0][0]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glEnable(GL_TEXTURE_2D);
    glShadeModel(GL_FLAT);
}

void CALLBACK display(void)
{
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 1.0);
    glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0);
    glVertex3f(0.0, 1.0, 0.0);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(0.0, -1.0, 0.0);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(1.0, -1.0, 0.0);
    glTexCoord2f(0.0, 1.0);
    glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0);
    glVertex3f(2.41421, 1.0, -1.41421);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(2.41421, -1.0,-1.41421);
    glEnd();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0*(GLfloat)w/(GLfloat)h, 1.0, 30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.6);
}

int main(int argc, char** argv)
{
```

```
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 10-1");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
     return 0;
}
```

棋盘纹理是由子程序 makeCheckImage()生成的，纹理映射的初始化在函数 myinit()中完成。高分辨率的纹理图由 glTexImage2D()指定，其参数说明了图像的大小、图像的类型、图像的位置和一些别的属性。关于该函数的详细说明参见下一节。

接下来的四个 glTexParameter*()函数指定纹理是如何进行粘贴的，以及当纹理与屏幕像素点之间颜色不匹配时如何对颜色进行过滤处理。然后，由函数 glTexEnv*()把绘图模式设置为 GL_DECAL，从而使用纹理图的颜色绘制出带纹理的多边形。最后，函数 glEnable()将纹理映射激活。

函数 display()绘制两个多边形。

注意：纹理坐标是与顶点坐标一起指定的。函数 glTexCoord*()的操作类似于用来设置当前纹理坐标的 glNormal()命令；在调用 glTexCoord*()语句之后，再次调用该命令之前，所有的顶点命令都使用该纹理坐标。

10.2　指定纹理

函数 glTexImage2D()定义了一个二维纹理。下面对该函数进行说明。

---

void glTexImage2D(GLenum target, GLint level, GLint components, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *pixels);

定义一个二维纹理。参数 target 是为 OpenGL 的升级准备的参数，目前，target 参数必须设置成 GL_TEXTURE_2D。如果提供了多种分辨率的纹理图，则要用到参数 level；当纹理图只有一种分辨率时，level 为 0。

参数 components 是一个 1 至 4 之间的整数，说明 R、G、B 和 A 分量中哪几个分量用于颜色的调整或融合。1 表示 R，2 表示 R 和 A，3 代表 R、G 和 B，而 4 表示 R、G、B 和 A。

参数 width 和 height 给出纹理图像的尺寸。参数 border 指明边界的宽度，通常为 0。无论是宽度还是高度都必须是 2^m+2^b 的形式，这里 m 是一个整数（对于宽度和高度可以取不

同的值)，b 为边界值。纹理图像的最大尺寸取决于具体的 OpenGL 实现，但是不能小于 64 × 64 (包括边界时，不小于 66 × 66)。如果将 width 或 height 置为零，则纹理映射无效。

参数 format 和 type 描述了纹理图像的数据格式和数据类型，和函数 glDrawPixels()中相应的两个参数的意义相同。实际上，纹理数据的格式与 glDrawPixels()函数使用的数据格式是相同的。

参数 format 可以是 GL_COLOR_INDEX，GL_RGB，GL_RGBA，GL_RED，GL_GREEN，GL_BLUE，GL_ALPHA，GL_LUMINANCE 或者是 GL_LUMINANCE_ALPHA，除了 GL_STENCIL_INDEX 和 GL_DEPTH_COMPONENT 之外，其他格式同样适用于函数 glDrawPixels()。类似的，参数 type 可以是 GL_BYTE，GL_UNSIGNED_BYTE，GL_SHORT，GL_UNSIGNED_SHORT，GL_INT，GL_UNSIGNED_INT，GL_FLOAT，或 GL_BITMAP。

最后，参数 pixels 中包含了纹理图像数据。该数据描述了纹理图像及其边界。

在下一节中，将对参数 border 和 level 做更细致的描述。参数 level 定义了不同分辨率的纹理，自然就引出了在使用纹理时对其进行过滤处理的问题。

10.2.1 使用纹理边界

如果要使用比 OpenGL 实现所允许的范围大得多的纹理图，稍加处理就可以使几个纹理粘贴到一起，生成一个较大的纹理。比如说，如果需要把一个两倍于最大允许尺寸的纹理映射到一个正方形上，则可以首先把这个正方形分成四个小正方形，然后在绘制每一个小正方形之前装载不同的纹理。

由于同一时刻只能使用一个纹理图，这就可能在纹理的边界处产生一些问题，尤其在进行某种形式的线性过滤的时候。用于边界像素点处的纹理值必须与边界之外的相邻纹理值做一个平均。如果给每个纹理定义一个边界，这个边界上的纹理单元与纹理图上相邻的纹理单元的值相等，在线性平均后，就可以得到正确的结果。为了正确地实现这一点，应注意到每个纹理图可能有八个相邻的区域。

如果纹理只覆盖了图元的一部分，那么纹理边界颜色也需要进行处理。

10.2.2 细节等级（Level of Details）

像场景中的其他物体一样，可以在离观察点不同距离的地方观察带有纹理的物体。例如，在一个动态场景中，当一个带纹理的物体离开观察点时，该纹理图要随透视图的变小而变小。为了实现这个目标， OpenGL 将纹理图过滤成一个合适的尺寸粘贴到物体上，同时不会在视觉上产生令人讨厌的人工雕琢的感觉。为了避免这种人工修饰的痕迹，程序员可以指定一系列预先过滤的，且分辨率减小的纹理图，称为 mipmaps，如图 10-3 所示。然后，OpenGL 按照物体的尺寸（以像素为单位）自动地确定应该使用哪个纹理图。利用这种方法，纹理图上的细节等级对绘制在屏幕上的图像来说是十分合适的——当物体的图像变小时，纹理图的尺寸也变小。

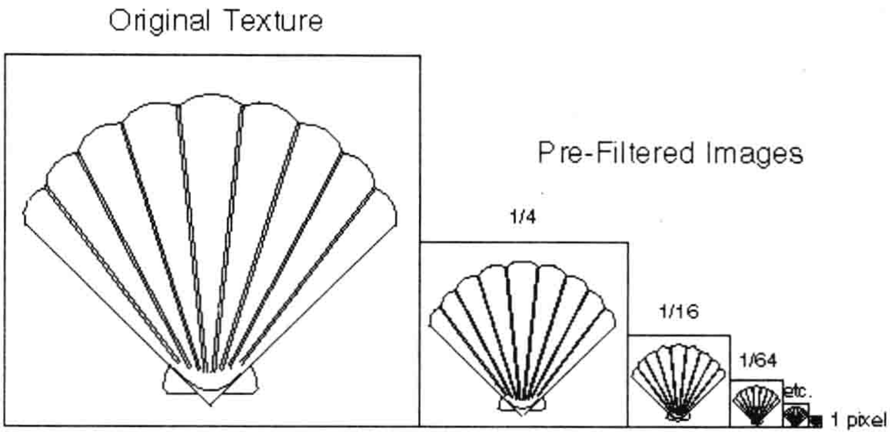


图 10-3　Mipmaps

上述技术称为 mipmapping（Mip 代表拉丁文中的 multim im parvo，意思是“在狭窄地方里的许多东西”）。

要使用 mipmapping，程序员必须按 2 次幂的方式给出从 1×1 大小到最大尺寸图之间的所有尺寸的纹理图。例如，如果最高分辨率纹理是 64×16 的话，程序员就必须给出 32×8，16×4，8×2，4×1，2×1 和 1×1 大小的纹理。较小的纹理一般要进行过滤处理。OpenGL 不要求任何特殊方法来计算较小纹理图，因此尺寸不同的纹理图之间是完全不相关的。

为了指定这些纹理，对于不同的纹理均需调用 glTexImage2D()函数。参数 level 从零开始，确定了在序列中指定的是哪一个纹理。对于上一个例子，64×16 的最大纹理图的 level 值为 0，32×8 的纹理图 level 值为 1，如此类推。此外，为了使经过 mipmapping 处理后的纹理起作用，还必须采取一种恰当的过滤方法。

例 10-2 说明从 32×32 递减到 1×1 的六个纹理图的用法。该程序绘制了一个矩形，这个矩形从前景开始，直到最终消失，如彩图 20 所示。注意，需要使用该纹理图的 64 个拷贝粘贴到这个矩形上，每个方向上有 8 个纹理图。为了说明纹理图是如何相互连接的，在程序中纹理图之间的颜色互不相同。

例 10-2　Mipmapping 技术范例：mipmap.c

```
#include "windows.h"


#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>


GLubyte mipmapImage32[32][32][3];
GLubyte mipmapImage16[16][16][3];
GLubyte mipmapImage8[8][8][3];
```

```
GLubyte mipmapImage4[4][4][3];
GLubyte mipmapImage2[2][2][3];
GLubyte mipmapImage1[1][1][3];

void CALLBACK loadImages(void)
{
    int i, j;
    for (i = 0; i < 32; i++)
    {
        for (j = 0; j < 32; j++)
        {
            mipmapImage32[i][j][0] = 255;
            mipmapImage32[i][j][1] = 255;
            mipmapImage32[i][j][2] = 0;
        }
    }
    for (i = 0; i < 16; i++)
    {
        for (j = 0; j < 16; j++)
        {
          mipmapImage16[i][j][0] = 255;
          mipmapImage16[i][j][1] = 0;
          mipmapImage16[i][j][2] = 255;
        }
    }
    for (i = 0; i < 8; i++)
    {
        for (j = 0; j < 8; j++)
        {
          mipmapImage8[i][j][0] = 255;
          mipmapImage8[i][j][1] = 0;
          mipmapImage8[i][j][2] = 0;
        }
    }
    for (i = 0; i < 4; i++)
```

```
    {
        for (j = 0; j < 4; j++)
        {
          mipmapImage4[i][j][0] = 0;
          mipmapImage4[i][j][1] = 255;
          mipmapImage4[i][j][2] = 0;
        }
    }
    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
        {
            mipmapImage2[i][j][0] = 0;
            mipmapImage2[i][j][1] = 0;
            mipmapImage2[i][j][2] = 255;
        }
    }
    mipmapImage1[0][0][0] = 255;
    mipmapImage1[0][0][1] = 255;
    mipmapImage1[0][0][2] = 255;
}

void CALLBACK myinit(void)
{
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glShadeModel(GL_FLAT);
    glTranslatef(0.0, 0.0, -3.6);
    loadImages();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexImage2D(GL_TEXTURE_2D, 0, 3, 32, 32, 0,
                 GL_RGB, GL_UNSIGNED_BYTE,
                 &mipmapImage32[0][0][0]);
    glTexImage2D(GL_TEXTURE_2D, 1, 3, 16, 16, 0,
                 GL_RGB, GL_UNSIGNED_BYTE,
```

```
                &mipmapImage16[0][0][0]);
    glTexImage2D(GL_TEXTURE_2D, 2, 3, 8, 8, 0,
                GL_RGB, GL_UNSIGNED_BYTE,
                &mipmapImage8[0][0][0]);
    glTexImage2D(GL_TEXTURE_2D, 3, 3, 4, 4, 0,
                GL_RGB, GL_UNSIGNED_BYTE,
                &mipmapImage4[0][0][0]);
    glTexImage2D(GL_TEXTURE_2D, 4, 3, 2, 2, 0,
                GL_RGB, GL_UNSIGNED_BYTE,
                &mipmapImage2[0][0][0]);
    glTexImage2D(GL_TEXTURE_2D, 5, 3, 1, 1, 0,
                GL_RGB, GL_UNSIGNED_BYTE,
                &mipmapImage1[0][0][0]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,  GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,  GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,  GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
            GL_NEAREST_MIPMAP_NEAREST);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glEnable(GL_TEXTURE_2D);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 8.0);
    glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(8.0, 8.0);
    glVertex3f(2000.0, 1.0, -6000.0);
    glTexCoord2f(8.0, 0.0);
    glVertex3f(2000.0, -1.0, -6000.0);
    glEnd();
```

```
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0*(GLfloat)w/(GLfloat)h, 1.0,  30000.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 10-2");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return 0;
}
```

例 10-2 的运行结果如图 10-4 所示。

图 10-4　例 10-2 的运行结果

例 10-2 说明了 mipmapping 过程：将每个 mipmap 设置成不同的颜色，从而可以在一个纹理图代替另一个时明显地看到这一过程。实际上，定义 mipmaps 是为了使过渡尽可能地平滑。因此，低分辨率的纹理图通常是最初高分辨率纹理图过滤后的翻版。构造一系列这样的 mipmaps 实际上是一个软件处理过程，因而并不是 OpenGL 的一部分。然而， mipmap 的构造是一个非常重要的操作，因此在 OpenGL 实用库中包含了三个子程序，以支持用作纹理的图像操作。

假定已经构造了 level 为零的最高分辨率纹理图，子程序 gluBuild1DMipmaps()和 gluBuild2DMipmaps()将定义和构造出直到 1×1 分辨率的一系列 mipmaps。这两个子程序都要求原始图像适合于用作纹理，即其尺寸是 2 的幂次方。由于大多数扫描的图像都不满足这个条件，因此程序员需将输入图像缩放到适当的尺寸。GLU 库提供了一个子程序 gluScaleImage()来完成这一操作。

---

int gluBuild1DMipmaps(GLenum target, GLint components, GLint width, GLenum format, GLenum type, void *data);
int gluBuild2DMipmaps(GLenum target, GLint components, GLint width, GLint height, GLenum format, GLenum type, void *data);

构造一系列 mipmaps。参数 target, components, width, height, format, type 和 data 与函数 glTexImage1D()和 glTexImage2D()中的参数是完全一样的。如果所有的 mipmaps 构造成功，函数的返回值为 0，否则，返回 GLU 错误代码。

---

int gluScaleImage(GLenum format, GLint widthin, GLint heightin, GLenum typein, const void *datain, GLint widthout, GLint heightout, GLenum typeout, void *dataout);

使用适当的像素存储模式对图像进行缩放。从文件 datain 中打开数据，参数 format, typein 和 typeout 指的是函数 glDrawPixels()所支持的任意一种数据类型或格式。通过线性插值和过滤操作（从 widthinXheightin 指定的尺寸转换到 widthoutXheightout 指定的尺寸）对图像进行缩放，并将结果图像写到缓冲区 dataout 中。函数执行成功时返回 0，失败时返回一个 GLU 错误代码。

10.2.3 控制过滤

纹理图是正方形或矩形的，但是，在映射到多边形或物体表面，转换为屏幕坐标后，纹理图中的单个纹理单元很少能够与最终屏幕图像中的单个纹理单元一一对应。根据所使用的变换和纹理映射，屏幕上的一个像素点可以与任何大小的事物对应，从一个纹理单元中的一

小部分（放大后），到许多纹理单元的集合（缩小后），如图 10-5 所示。不管在哪种情况下，应该使用哪一个纹理单元值以及如何对这些单元进行平均或插值，是无法确切地知道的。这样，OpenGL 就程序员指定的某一种过滤方法来决定这些计算，以便在速度和图像质量之间进行适当的折衷。

另外，还可以分别为放大和缩小指定不同的过滤方式。

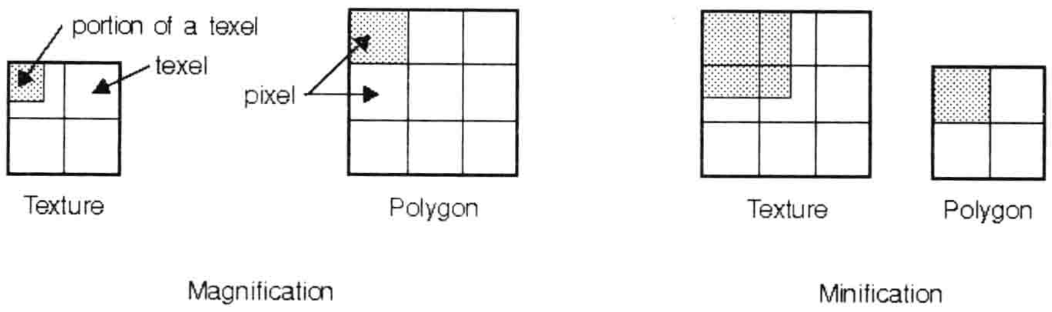


图 10-5　纹理的缩放

在某些情况下，是进行放大还是缩小操作并不十分清楚。如果需要在 x 和 y 两个方向上同时拉伸（或缩小）mipmap 图，那么则需要进行放大（或缩小）。然而，如果在一个方向上拉伸而在另一个方向上缩小，这时，OpenGL 则在放大和缩小之间进行选择以便在多数情形下能得到尽可能最好的效果。当然，最好的方法是使用映射中不发生扭曲的纹理坐标来避免这种情况。

下面的代码列举了用函数 glTexParameter*()指定放大和缩小的过滤方法。

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

函数 glTexParameter*()的第一个参数或者是 GL_TEXTURE_2D，或者是 GL_TEXTURE_1D，这取决于所使用的是二维还是一维纹理。为了进行放大或缩小，对应的第二个参数是 GL_TEXTURE_MAG_FILTER，或者是 GL_TEXTURE_MIN_FILTER。第三个参数指定过滤方法；表 10-1 列出了可能的值。

表 10-1　用于放大和缩小的过滤方法

参数	值
GL_TEXTURE_MAG_FILTER	GL_NEAREST or GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, or GL_LINEAR_MIPMAP_LINEAR

如果选择 GL_NEAREST，则坐标最靠近像素点中心的纹理单元既用于放大过滤又用于缩小过滤。这可能导致锯齿现象。如果选择参数 GL_LINEAR，则使用位于像素点中心最近处的 2×2 纹理单元列的加权线性平均，也是既用于放大又用于缩小。当纹理坐标靠近纹理图的边缘时，邻近的 2×2 纹理单元列有可能包含一些纹理图之外的点。在这些情形下，实际使用的纹理值不但取决于生效的是 GL_REPEAT 还是 GL_CLAMP，还取决于程序员是否给该纹理指定了边界。GL_NEAREST 比 GL_LINEAR 的计算量要少，因此执行速度较快，但是 GL_LINEAR 可以提供更平滑的效果。

如果进行放大，即使提供了 mipmaps，通常还是要用到最大的纹理图（level = 0）。如果是缩小，则可以选择使用最适当的一种或两种 mipmaps 过滤方法。

10.2.4 一维纹理

有时使用一维纹理就足够了——例如，画一条带纹理的带子，这里所有的变化都是在一个方向上的。一维纹理相当于 height = 1 的二维纹理，只是在纹理顶部和底部没有边界。用函数 glTexImage1D()可以指定一个一维纹理。

---

void glTexImage1D(GLenum target, GLint level, GLint components, GLsizei width, GLint border, GLenum format, GLenum type, const GLvoid *pixels);

定义一维纹理。除了参数 image 在这里是一维的纹理单元数组之外，其他所有的参数都与函数 glTexImage2D()中使用的参数意义相同。如前所述，参数 width 的取值必须是 2 的幂次方（2^m，有一个边框时为 2^m+2）。一维纹理也可以提供 mipmaps，也可以使用同样的过滤方法。

10.3 调节和混合

在上述例子中，纹理图的值直接用于表面渲染。当然也可以使用这些值来调节不带纹理的表面的颜色，或者将纹理颜色与不带纹理的物体表面颜色相融合。通过为函数 glTexEnv*() 提供合适的参数，就可以实现这三个纹理映射功能中的任意一个。

---

void glTexEnv{if}{v}(GLenum target, GLenum pname, TYPEparam);

设置当前纹理映射功能。参数 target 必须是 GL_TEXTURE_ENV。如果参数 pname 是 GL_TEXTURE_ENV_MODE，那么参数 param 可以取 GL_DECAL，GL_MODULATE，或者 GL_BLEND，从而指定纹理值是如何与所处理的片原的颜色值相结合的。在 decal 模式下，纹理颜色将取代片原的颜色。在其他两种模式下，或使用 RGBA 纹理时，最终的颜色

是纹理颜色和片原颜色的合成。如果参数 pname 为 GL_TEXTURE_ENV_COLOR，则参数 pname 是一个包含四个浮点数的数组，四个元素分别代表 R、G、B 和 A 四个分量。只有在指定了 GL_BLEND 的纹理功能时，这些值才被使用。

纹理映射功能针对所选的纹理成分和不带纹理的颜色进行操作。在用函数 glTexImage*d() 来指定纹理图时，第三个参数是为每个纹理单元所选的 R、G、B 和 A 分量的个数。如只选一个分量，则表示亮度值（L）；如果选定两个分量，那么第一个是亮度，第二个是一个 alpha 值（A）。三个分量组成一个 RGB 颜色的三元组（C），而四个分量则规定了一个 RGB 三元组和一个 alpha 值。注意，这种选择是在应用像素转换功能之后执行的。纹理映射功能所用的成分如表 10-2 所示。

表 10-2　贴花、调节及融合功能

成分（分量）	贴花方式	调节方式	融合方式
1	未定义	$C = L_tC_f$, $A = A_f$	$C = (1-L_t)C_f + L_tC_c$, $A = A_f$
2	未定义	$C = L_tC_f$, $A = A_tA_f$	$C = (1-L_t)C_f + L_tC_c$, $A = A_tA_f$
3	$C = C_t$, $A = A_f$	$C = C_tC_f$, $A = A_f$	未定义
4	$C = (1_t)C_f + A_tC_t$, $A = A_f$	$C = C_tC_f$, $A = A_tA_f$	未定义

注意：在表 10-2 中，下标 t 代表一个纹理值，f 代表一个片原值，c 代表由 GL_TEXTURE_ENV_COLOR 所分配的值，没有下标的项代表最后的计算值。另外，表中的一个颜色三元组与一个标量相乘表示各分量分别与该标量相乘。两个颜色三元组相乘（或相加）表示第二个三元组的各分量与第一个三元组的对应分量分别相乘（或相加）。

从表 10-2 中可以看到，只有颜色分量的个数为 3 或 4 时，贴花模式才有意义（注意，在颜色索引模式下不使用纹理映射）。

颜色调节是指用纹理图的颜色来调节片原的颜色。对一个或两个分量，以相同的值与这些颜色值相乘，使纹理图在片原颜色（如果亮度为 1）和黑色（如果亮度为 0）之间进行调节。使用三个或四个分量时，则是各个输入颜色分量都要与纹理图中对应的颜色值相乘。如果有一个为 alpha 值，那么该 alpha 值要与片原的 alpha 值相乘。利用调节功能，可以创建一种与光照环境对应的纹理。

融合方式只在一个或两个分量时才有意义。亮度的使用类似于 alpha 值，它将片原的颜色与由 GL_TEXTURE_ENV_COLOR 指定的颜色进行融合。在两个分量的情况下，片原的 alpha 也要与纹理的 alpha 值相乘。

10.4 分配纹理坐标

在绘制带纹理的场景时，必须同时提供每一个顶点的物体坐标和纹理坐标。在变换之后，由物体坐标来决定在屏幕上什么地方对特定顶点进行渲染，而由纹理坐标来决定将纹理图中哪一个纹理单元分配给该顶点。与在多边形或线段的两个顶点之间对颜色进行线性插值一样，纹理坐标也是在顶点之间进行线形插值。

纹理坐标可以由一个、两个、三个或四个坐标组成，通常以字母 s，t，r 和 q 表示，以区别于物体坐标（x, y, z 和 w）及求值程序坐标（u 和 v）。对于一维纹理，使用 s 坐标；对于二维纹理，使用 s 和 t 坐标。目前，r 坐标被忽略。与 w 一样， q 坐标一般取为 1，它用于创建齐次坐标。确定纹理坐标的函数 glTexCoord*()与函数 glColor*()、glVertex*()和 glNormal*()类似——它们具有类似的变量，都用在函数 glBegin()与 glEnd()之间。通常，纹理坐标的值在 0 到 1 之间。

---

void glTexCoord{1234}{sifd}{v}(TYPE coords);

设置当前纹理坐标(s, t, r, q)。接下来调用 glVertex*()函数将当前纹理坐标分配给顶点。函数 glTexCoord1*()将 s 坐标设定为指定值，将 t 和 r 设置为 0，q 为 1；函数 glTexCoord2*()将 s 和 t 坐标设置为指定值，将 r 设置为 0，q 为 1；函数 glTexCoord3*()将 s、t 和 r 坐标设置为指定值，q 为 1；函数 glTexCoord4*()用于设置 4 个坐标值。利用合适的后缀（s, i, f 或 d）及相应的 TYPE 类型值 (GLshort, GLint, Glfloat 或 GLdouble)可以指定坐标的数据类型。可以分别设置坐标的各个分量，也可以利用向量来设置坐标值。在纹理映射之前，要用一个 4 阶方阵与纹理坐标相对应。

下一节将讨论如何计算纹理坐标。将纹理坐标作为顶点坐标的一个函数时，OpenGL 可以自动计算出纹理坐标。

10.4.1 计算纹理坐标

二维纹理是方形或矩形的图像。在最简单的情况下，人们把一个矩形纹理映射到矩形的模型上。

我们考虑这样一种情形：假设墙壁的高度是宽度的三分之二，而纹理仍然是方形的。为了避免纹理扭曲，我们必须将墙壁与部分纹理图映射起来。假设将底部的三分之二纹理贴到墙壁上。在这种情况下，由于墙壁顶点是逆时针旋转的，因此坐标（0,0），（1,0），（1,2/3）和（0,2/3）作为纹理坐标。

只有诸如锥体和圆柱体等很少的一些曲面在映射到一个平面时才不发生几何扭曲，而其他任何形状都要发生扭曲。一般地，表面的曲度越大，纹理扭曲得越厉害。

图 10-6 显示了两个圆环，其中一个圆环直径较小（扭曲较严重），而另一个直径较大（扭曲比较小）。

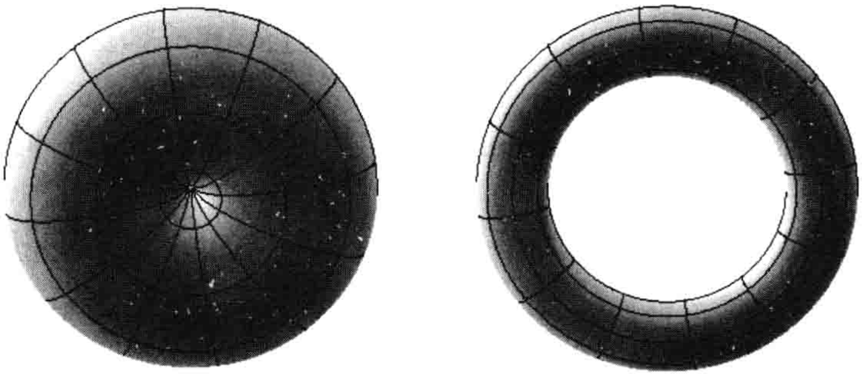


图 10-6　纹理图的扭曲

如果将纹理粘贴到由求值程序产生的样条曲面上，则物面参数 u 和 v 有时可作为纹理坐标使用。一般地，要成功地将纹理映射到由多边形近似的表面上，更需要艺术家的想象力。

10.4.2　纹理的重复和截取

可以将纹理坐标超出区间[0,1]之外，在纹理图中对这些坐标进行截取或重复。

在例 10-1 中，如果将正方形的纹理坐标映射到 0.0 和 3.0 之间，其结果则如图 10-7 所示。

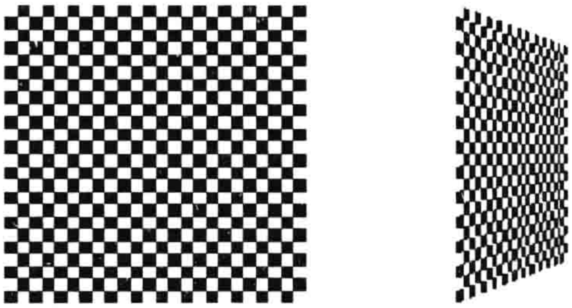


图 10-7　纹理的重复

```
glBegin(GL_POLYGON);
  glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
  glTexCoord2f(0.0, 3.0); glVertex3f(-2.0, 1.0, 0.0);
  glTexCoord2f(3.0, 3.0); glVertex3f(0.0, 1.0, 0.0);
  glTexCoord2f(3.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
glEnd();
glBegin(GL_POLYGON);
```

```
    glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
    glTexCoord2f(0.0, 3.0); glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(3.0, 3.0); glVertex3f(2.41421, 1.0, -1.41421);
    glTexCoord2f(3.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421)
 glEnd();
```

在本例中，由于都是对函数 glTexParameter*()进行调用，因此纹理 同时在 s 和 t 两个方向进行重复。

```
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_S_WRAP, GL_REPEAT);
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_T_WRAP, GL_REPEAT);
```

如果函数的最后一个参数是 GL_CLAMP，而不是 GL_REPEAT，其结果将如图 10-8 所示。

图 10-8 对纹理进行截取

注意，边框颜色在纹理图之外是连续的。当然也可以在一个方向上进行截取，而在另一个方向上进行重复，这样产生的结果如图 10-9 所示。

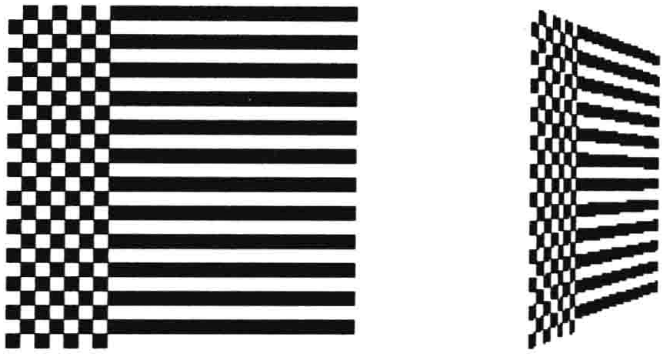


图 10-9 对纹理进行重复和截取

上面介绍了函数 glTexParameter*()的所有可能的参数，这里对它们进行一下总结。

---

void glTexParameter{if}{v}(GLenum target, GLenum pname, TYPE param);

该函数设置各种控制参数。参数 target 可以是 GL_TEXTURE_2D，或 GL_TEXTURE_1D，分别表示二维或一维纹理。参数 pname 和 param 的可能取值如表 10-3 所示。可以用向量的形式或独立的数值为 GL_TEXTURE_BORDER_COLOR 指定参数。如果是整型量，则要根据表 5-1 把它们转换成浮点数，同时将其截取到区间[0，1]之间。

表 10-3　函数 glTexParameter*()的参数

参数	取值
GL_TEXTURE_WRAP_S	GL_CLAMP, GL_REPEAT
GL_TEXTURE_WRAP_T	GL_CLAMP, GL_REPEAT
GL_TEXTURE_MAG_FILTER	GL_NEAREST, GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEARES T, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR
GL_TEXTURE_BORDER_COLOR	区间[0，1]之间的任意四个数

10.5　自动纹理坐标生成

利用纹理映射可以在模型上制作轮廓，或者模拟对环境的反射。为了得到这种效果，可以让 OpenGL 自动生成纹理坐标，而不必用函数 glTexCoord*()明确地指定这些坐标。实际上，对于诸如显示轮廓和反射环境等众多不同的应用，自动纹理坐标生成机制都相当有用。函数 glTexGen()用于自动生成纹理坐标。

---

void glTexGen{ifd}{v}(GLenum coord, GLenum pname, TYPEparam);

自动生成纹理坐标的函数。第一个参数 coord 必须是 GL_S，GL_T，GL_R，或 GL_Q，表明是否要生成纹理坐标 s，t，r 或 q。参数 pname 为 GL_TEXTURE_GEN_MODE，GL_OBJECT_PLANE 或 GL_EYE_PLANE，如果为 GL_TEXTURE_GEN_MODE，则 param 是一个整型量（在该命令的向量形式中为指向一个整型量的指针），该整型量可以是 GL_OBJECT_LINEAR，GL_EYE_LINEAR，或 GL_SPHERE_MAP。这些符号常数决定了用哪一个函数来生成纹理坐标。对于参数 pname 的其他可能值，pname 既可以是一个指向数组的指针，也可以是一个指定纹理生成函数参数的数值。

10.5.1 创建轮廓

当参数为 GL_TEXTURE_GEN_MODE 和 GL_OBJECT_LINEAR 时，其纹理生成函数是物体顶点坐标(x_0,y_0,z_0,w_0)的线性组合：

生成的坐标= $p1x_0 + p2y_0 + p3z_0 + p4w_0$

当参数 pname 为 GL_OBJECT_PLANE 时，p1, ..., p4 就是函数 glTexGen*()的参数 param 的值。在 p1, ..., p4 被正确地规一化后，该函数给出从该顶点到一个平面的距离。例如，如果 p1 = p3 = p4 = 0 而 p2 = 1，该函数给出了顶点到平面 y = 0 的距离。在平面的一侧，距离值为正值，而在平面的另一侧是负值，当顶点位于该平面时，距离为零。

在例 10-3 中，茶壶上绘制了等间距的轮廓线，这些线显示了与平面 y = 0 的距离。由于只显示一个属性（与平面的距离），所以一维的纹理图就够了。除了在等间距的间隔内有红色标记外，纹理图全是绿色。由于该茶壶位于 x-y 平面上，因此所有的轮廓线都垂直于其基平面。图 10-10 显示了该程序的最终结果。

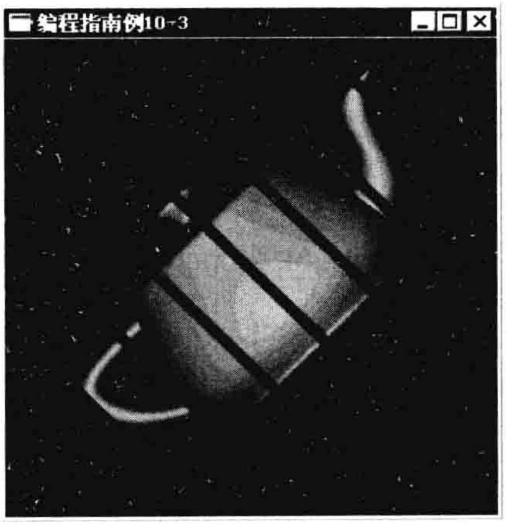


图 10-10 例 10-3 的运行结果

例 10-3 自动纹理坐标生成

```
#include <windows.h>

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

#define   stripeImageWidth 32
GLubyte   stripeImage[3*stripeImageWidth];
```

```
void CALLBACK loadStripeImage(void)
{
    int j;
    for (j = 0; j < stripeImageWidth; j++)
    {
      stripeImage[3*j] = (j<=4) ? 255 : 0;
      stripeImage[3*j+1] = (j>4) ? 255 : 0;
      stripeImage[3*j+2] = 0;
    }
}

GLfloat sgenparams[] = {1.0, 0.0, 0.0, 0.0};

void CALLBACK myinit(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    loadStripeImage();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexImage1D(GL_TEXTURE_1D, 0, 3, stripeImageWidth, 0,
                 GL_RGB, GL_UNSIGNED_BYTE, stripeImage);
    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
    glTexGenfv(GL_S, GL_OBJECT_PLANE, sgenparams);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_1D);
    glEnable(GL_CULL_FACE);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
```

```
    glFrontFace(GL_CW);
    glCullFace(GL_BACK);
    glMaterialf (GL_FRONT, GL_SHININESS, 64.0);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
    glRotatef(45.0, 0.0, 0.0, 1.0);
    auxSolidTeapot(2.0);
    glPopMatrix ();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
             glOrtho (-3.5, 3.5, -3.5*(GLfloat)h/(GLfloat)w,
                      3.5*(GLfloat)h/(GLfloat)w, -3.5, 3.5);
    else
             glOrtho (-3.5*(GLfloat)w/(GLfloat)h,
                      3.5*(GLfloat)w/(GLfloat)h, -3.5, 3.5, -3.5, 3.5);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
   auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
   auxInitPosition (0, 0, 300, 300);
   auxInitWindow ("编程指南例 10-3");
```

```
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
     return 0;
}
```

如上所示，我们以参数 GL_TEXTURE_GEN_S 调用函数 glEnable()，来激活 s 坐标的纹理坐标生成。如果要生成其他坐标，则要使用 GL_TEXTURE_GEN_T，GL_TEXTURE_GEN_R，或 GL_TEXTURE_GEN_Q。注意，参数 GL_REPEAT 用于轮廓线纹理的重复。

如果将定义数组 sgenparams 这一行改为

GLfloat sgenparams[] = {1, 1, 1, 0};

那么所有的轮廓线将平行于平面 $x + y + z = 0$，如彩图 18b 所示。

参数 GL_OBJECT_LINEAR 用于计算模型坐标系中的纹理坐标。在例 10-3 中，轮廓线垂直于茶壶的基平面，无论茶壶怎么旋转或从哪一个角度观察，这一特性都不变。有时可能希望在人眼坐标系中计算轮廓线，也就是说，在计算到平面的距离之前，先用模型视图矩阵的逆矩阵乘以向量(p1 p2 p3 p4)。如果在指定 GL_TEXTURE_GEN_MODE 时指定了 GL_EYE_LINEAR，就能够得到正确的结果。利用下面的公式即可以生成纹理坐标：

生成的坐标= $p1'x_e + p2'y_e + p3'z_e + p4'w_e$

其中，$(p1'\ p2'\ p3'\ p4') = (p1\ p2\ p3\ p4)^{M^{-1}}$。

在本例中，(x_e, y_e, z_e, w_e)是顶点的人眼坐标，而 p1, ..., p4 则是在 pname 为 GL_EYE_PLANE 时函数 glTexGen*()的参数 param 的值。如果要看一看上式的效果，可以将上一个例子中的参数 sgenparams 改回{1, 0, 0, 0}，同时将 GL_OBJECT_LINEAR 改为 GL_EYE_LINEAR。其结果如彩图 18c 所示。

在上述的例子中，都是用单个纹理坐标来产生轮廓线。然而纹理坐标 s 和 t 是可以单独生成的，这表示与两个不同平面的不同距离。如果适当地构造一个二维纹理，就可以同时看到两列轮廓线。

10.5.2 环境映射

环境映射的目的是以该物体周围的环境颜色渲染物体。例如，当我们在一个房间里观察一个抛光很好且具有极佳反射性的银制品时，观察者将会发现银制品能将周围的墙壁、地板和其他物体反射出来。所看到的反射物体取决于眼睛的位置、银制品的位置和该银制品表面的摆放角度。要实现环境映射，程序员要做的事情就是创建一个合适的纹理图，然后让 OpenGL 生成纹理坐标即可。

环境映射是在环境中物体相对于光洁表面距离很远这样的假设下所做的近似。根据这一假设，我们选取从眼睛到物体表面的光线，并使该光线从表面上反射回来，这样就可以得到物体表面上某一点的颜色。反射光线的方向完全决定了该点的颜色。对一个平面纹理

图的各个方向进行颜色编码，等价于在环境的中间放置一个抛光的理想的球体，然后用一个长焦距镜头的照相机在远处拍摄球体的照片。从数学上讲，焦距可以无限长，照相机也可以在无穷远处。因此，编码就包括了纹理图的圆形区域，并且与图的顶部、底部、左侧和右侧边缘相切。

要制作完全正确的环境纹理图，必须用一个很大的银球，使用一个焦距长度为无穷大，且放置在无穷远处的相机拍摄下一张该球体在某种环境下的照片，并将该照片扫描出来。实际上，可以将广角镜（或鱼眼）相机拍摄的照片扫描出来。彩图 21a 显示了一幅环境映射纹理图。

创建出用于环境映射的纹理后，还要激活 OpenGL 的环境映射的算法。该算法要找出球面上与所渲染物体上有相同切面的点，同时将球体表面的对应点以可见的颜色绘制出来。如果要编写程序，所要做的就是将例 10-3 中这几行：

```
glTexGenfv(GL_S, GL_OBJECT_LINEAR, sgenparams);
glEnable(GL_TEXTURE_GEN_S);
```

改为如下语句：

```
glTexGenfv(GL_S, GL_SPHERE_MAP, 0);
glTexGenfv(GL_T, GL_SPHERE_MAP, 0);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

常数 GL_SPHERE_MAP 用来激活环境映射。在程序中，程序员需要同时指定 s 和 t 两个方向的 GL_SPHERE_MAP。

10.6 高级特性

本节详细讨论了纹理矩阵堆栈的操作及纹理坐标 q 的用法。在一般的纹理映射应用中通常不使用这两项高级技术。

10.6.1 纹理矩阵堆栈

正如在渲染之前用矩阵对模型坐标进行变换那样，在纹理映射之前，也需用一个 4 阶方阵对纹理坐标进行变换。在缺省情况下，该纹理矩阵是一个单位阵，这样一来，无论是直接指定的纹理坐标还是自动生成的纹理坐标都保持不变。然而在重画一个物体时，通过修改纹理矩阵就可以使纹理在物体的表面上产生滑动、旋转、拉伸或收缩。事实上，由于纹理矩阵纯粹是一个普通的 4 阶方阵，因此一样可以实现诸如透视这样的效果。

纹理矩阵与四个纹理坐标(s, t, r, q)相乘时，得到的向量(s' t' r' q')是齐次纹理坐标。换句话说，用 s'/q' 和 t'/q' 就可以对纹理图进行索引（目前不使用 r'/q'）。纹理矩阵实际上是深度至少为 2 的堆栈的栈顶矩阵。所有的标准矩阵操作函数，如 glPushMatrix()，glPopMatrix()，

glMultMatrix()和 glRotate()都可用于纹理矩阵。要修改当前的纹理矩阵，要像下面这样将矩阵模式设置成 GL_TEXTURE：

```
glMatrixMode(GL_TEXTURE); /* 进入纹理矩阵模式 */
glRotate(...);  /* ... 其他矩阵操作 ... */
glMatrixMode(GL_MODELVIEW); /* 返回模型视图模式 */
```

10.6.2　纹理的 q 坐标

一般四维纹理坐标中的 q 坐标的数学含义在前一节中已经作过说明。在投影或透视变换较多的情况下，可以利用坐标 q。

例如，假设要建立一个聚光灯模型，该聚光灯是不均匀的——可能中间稍亮，也可能不是圆形的。我们制作一幅与灯形状和强度相对应的纹理图，然后使用投影变换将其投影到该表面上，这样就可以模仿这种聚光灯在表面上的光亮效果。由于灯光可能照射到与之不垂直的表面上，因此，需要用投影变换（q*1）将灯锥投影到场景中的表面上。当观察者从不同的透视点观察场景时，还需要进行第二次投影变换。请参阅彩图 27。

如果所用的纹理图本身就是以透视方式拍摄的，也会发生类似的情况。与聚光灯一样，最终的视图取决于两次投影变换的组合。

第 11 章　帧缓存

本章主题

几乎所有图形程序的一个重要目标就是进行屏幕绘图。屏幕由像素的矩形阵列组成，在图像中每个像素点可以显示一小块颜色。为了画出这些像素，需要知道这些像素的颜色，这些信息存储在颜色缓存中。只要数据为每一个像素均匀存储，所有像素的存储就称为缓存。不同的缓存包含了每个像素的不同数据，一旦给定了缓存，则给每个像素分配的数据量就是相等的。存储像素的一位信息的缓存称为位平面。

如图 11-1 所示，OpenGL 窗口左下角的像素是（0，0），对应着该像素占据的 1×1 区域的左下角窗口坐标。一般情况下，像素(x, y)所填充的区域四个角点为（x,y,x+1,y+1）。

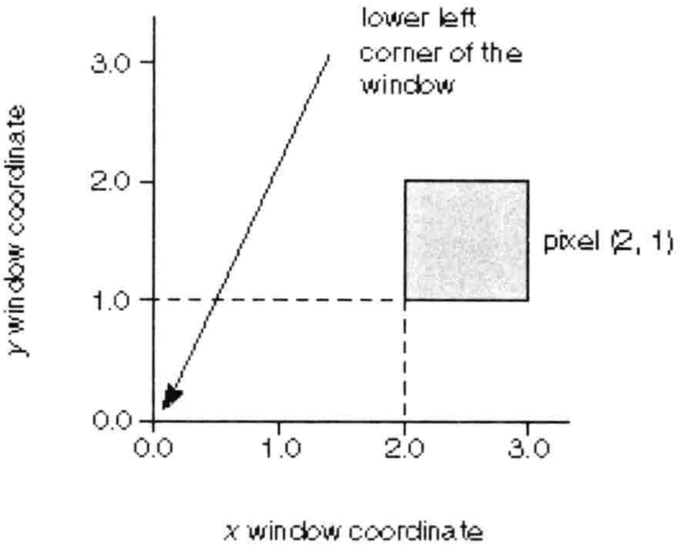


图 11-1　一个像素所占据的区域

作为一个缓存的例子，我们先来详细地研究一下颜色缓存。颜色缓存包含了将要在屏幕上显示的颜色信息。假定屏幕分辨率为 1280×1024，24 位真彩色——就是在屏幕上可以显示 2^{24}（=16,777,216）种颜色。由于 24 位可转换为 3 个字节（8 位/字节），因此，本例中的颜色缓存必须为屏幕上 1024×1280（=1,310,720）个像素中的每一个像素至少存储 3 个字节的数据。某种特定的硬件系统物理屏幕的像素可多可少，每个像素的颜色数据也不一样。然而，任何特定的缓存为每一个屏幕像素所保存的数据量都相等。

颜色缓存仅仅是保存像素信息的多个缓存之一。例如，在隐藏面的消除补救工具箱一节中，已经学习过保存像素深度信息的深度缓存。颜色缓存本身又由多个子缓存组成。系统的

帧缓存包括所有的这些缓存。除了颜色缓存之外，我们还没有直接接触过其他的缓存；但是实际上已经利用它们进行了诸如隐藏面删除、消除整个场景中的锯齿现象、模板化、绘制平滑运动以及其他的操作。

本章将描述在 OpenGL 实现中的所有缓存，以及如何使用它们。同时还要讨论在数据写入可视颜色缓存之前的一系列测试及像素操作。最后，解释了累积缓存的使用，累积缓存用来将绘制进颜色缓存的多幅图像累加到一起。本章包括如下几部分主要内容：

- 缓存及其使用，描述了所有可能用到的缓存及其用途，以及如何将这些缓存清除和激活用于写操作。
- 片原的测试及操作，解释了在计算出像素的位置和颜色之后，在这些信息被绘制到屏幕之前所进行的剪切、alpha、模板、以及深度缓存测试等操作。在用片原更新屏幕之前，可能还要执行一些其他操作，如混合、抖动、逻辑操作等。
- 累积缓存，描述如何利用累积缓存实现几种高级技术。这些技术包括完整场景锯齿现象的消除，运动模糊的使用，以及照相视野的模拟等。

11.1　缓存及其使用

OpenGL 系统能够对以下几种缓存进行操作：

1）颜色缓存：包括左前、右前、左后、右后、以及任意数量的辅助颜色缓存。

2）深度缓存。

3）模板缓存。

4）累积缓存。

特定的 OpenGL 实现决定了哪些缓存是可用的，以及每个像素包括多少位数据。另外，可以有多种视频或窗口类型，它们具有不同的可用缓存。最低限度应该确保拥有一个用于 RGBA 模式的颜色缓存，以及相关的颜色分量不为 0 的模板、深度和累积缓存；同时还要保证拥有一个用于颜色索引模式的颜色缓存，以及相应的深度和模板缓存。表 11-1 列出了用来查询 OpenGL 系统关于特定视频系统下每个像素缓存信息的函数 glGetIntegerv()所用到的参数。如果程序员使用的是 X 窗口系统，则可能会用到 glXGetConfig()来查询视频系统；这种情况的详细讨论请参见附录 4 及库函数详解部分。

表 11-1　每个像素缓冲区存储内容的查询参数

参数	意义
GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS	在颜色缓存中 R, G, B, A 每个分量的位数
GL_DEPTH_BITS	深度缓存中每个像素的位数
GL_STENCIL_BITS	模板缓存中每个像素的位数

续表

参数	意义
GL_ACCUM_RED_BITS, GL_ACCUM_GREEN_BITS, GL_ACCUM_BLUE_BITS, GL_ACCUM_ALPHA_BITS	在加速缓存中 R, G, B, A 每个分量的位数

11.1.1 各种缓存概述

1. *颜色缓存*

颜色缓存通常就是程序员绘图所用的缓存。颜色缓存包含着颜色索引或 RGB 颜色数据，还包括 alpha 值。支持立体感视图的 OpenGL 实现具有用于左右立体图像的左右颜色缓存。如果不支持立体感，就只用到左颜色缓存。类似地，双缓存系统具有前后两个缓存。单缓存系统仅有前缓存。在每一个 OpenGL 实现中都必须提供左前颜色缓存。

OpenGL 还支持多至 4 个可选择的、不可显示的辅助颜色缓存。OpenGL 并不指定这些缓存的用法，用户可以自己定义和使用它们。例如，程序员可以利用这些辅助缓存保存经常使用的图像，这样，再次需要时就可以将其拷贝到颜色缓存中，而不必重新绘制这幅图像。对 glCopyPixels()的详细描述请参阅读、写、拷贝像素数据一节。

可以用参数 GL_STEREO 或 GL_DOUBLE_BUFFER 来调用函数 glGetBooleanv()，确定是否支持立体感（具有左右缓存）或双缓存（具有前后缓存）。为了确定应用了多少辅助缓存，可以 GL_AUX_BUFFERS 为参数调用函数 glGetIntegerv()。

2. *深度缓存*

深度缓存用来存储每个像素的深度值。深度通常是以距眼睛的距离来度量的，因此，具有较大深度缓存值的像素要被具有较小深度缓存值的像素覆盖掉。这是一个十分有用的约定，这样深度缓存的行为就可以像深度测试中描述的那样进行修改。深度缓存有时也称为 z 缓存，可以理解为，x 和 y 表示了屏幕上的水平和垂直位移，z 表示了垂直于屏幕的距离。

3. *模板缓存*

模板缓存的一种用法是将作图限制在屏幕的某些部分中进行，就像是油画制作中用到的纸板和喷漆罐那样。比如想绘制一幅透过挡风玻璃观察到的图像，就要在模板缓存中存储挡风玻璃形状的图像，然后再绘制整个场景。模板缓存可以使得透过挡风玻璃看不见的物体不被画出。因此，编制驾驶模拟的应用程序，就可以一次性地画出汽车内的仪表和其他部分，在汽车运动时，只更新车外的场景即可。

4. *累积缓存*

与 RGBA 模式中的颜色缓存类似，累积缓存包含着 RGBA 颜色数据（在颜色索引模式中使用累积缓存将产生不可预测的结果）。它通常用于将一系列图像累加成一幅最终的合成

图像。利用这种方法，可以完成类似于这样的操作：通过超量采样消除场景的锯齿现象，然后利用均匀采样生成最终绘制到颜色缓存的像素的颜色值。像素不能直接绘制到累积缓存中；累加操作常常是先在矩形块中进行的，然后再与颜色缓存进行数据交换。

11.1.2　清空缓存

在图形程序中，清空屏幕（或任何缓存）是一种典型的代价昂贵的操作。在一个 1280 ×1024 的显示器上，清屏命令将作用于上百万个像素。对于简单的图形应用程序，清屏操作比绘图所花的时间更多。如果要清除的不仅是颜色缓存，还有深度和模板缓存，那么将花费掉 3 倍的时间。

为了解决这个问题，有些计算机用专门的硬件来同时清除多个缓存。OpenGL 的清除命令就是为了有效地利用类似体系而构造的。首先，指定要写入缓存的清除值，然后，一次性地发出命令执行清除操作，传递需要清除的所有缓存的列表。如果硬件能够同时清除这些缓存，操作将同时完成；否则，按顺序清除各个缓存。

下面的命令为每个缓存设置清除值。

---

void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
void glClearIndex(GLfloat index);
void glClearDepth(GLclampd depth);
void glClearStencil(GLint s);
void glClearAccum(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);

这些命令为 RGBA 模式及颜色索引模式中颜色缓存、深度缓存、模板缓存和累积缓存指定当前的清除值。GLclampf 及 GLclampd 类型的取值被截取在 0.0 和 1.0 之间。缺省的深度清除值为 1.0；其他缺省清除值为 0.0。清除命令设置的值，在被相同命令再次调用改变之前一直有效。

在选择清除值之后，就可以调用 glClear()清除缓存了。

---

void glClear(GLbitfield mask);

此函数清除指定的缓存。mask 的值是 GL_COLOR_BUFFER_BIT，GL_DEPTH_BUFFER_BIT，GL_STENCIL_BUFFER_BIT 及 GL_ACCUM_BUFFER_BIT 等常数的按位逻辑或（OR），用于确定哪个缓存将被清除。GL_COLOR_BUFFER_BIT 清除的是 RGBA 颜色缓存还是颜色索引缓存，取决于当前的系统颜色模式。当清除颜色或颜色索引缓存时，所有可写的颜色缓存都将被清空（见下一节）。

11.1.3 为写操作选择颜色缓存

绘图操作的结果可以放入任何一个颜色缓存中：前、后、左前、左后、右前、右后、或者任意辅助缓存。程序员既可以选择单个缓存作为绘图对象，也可以将对象设置成多个缓存的并集。例如，在双缓存的应用中，将同样的背景绘制到后缓存和前缓存中；然后，只在后缓存中作图（绘图完成后，再交换缓存）。然而，在某些情况下，可以将图同时绘入前后缓存，将特定双缓存窗口的一部分看成是单缓存。选择待写缓存可以用 glDrawBuffer()命令。

---

void glDrawBuffer(GLenum mode);

为写或清除操作选择需要激活的缓存。mode 的值可以取下列值中的一个:

GL_FRONT
GL_BACK
GL_RIGHT
GL_LEFT
GL_FRONT_RIGHT
GL_FRONT_LEFT
GL_BACK_RIGHT
GL_BACK_LEFT
GL_AUXi
GL_FRONT_AND_BACK
GL_NONE

略去了 RIGHT 或 LEFT 的参数指的是左右缓存；类似地，没有参数 FRONT 或 BACK，则指前后缓存。在 GL_AUXi 中的 i 表明了特定的辅助缓存。

11.1.4 缓存屏蔽

OpenGL 在向激活的颜色、深度、及模板缓存中写入数据之前，需要按下列命令指定的掩码对这些数据进行屏蔽操作。这些掩码要与对应的待写数据进行按位逻辑与（AND）操作。

---

void glIndexMask(GLuint mask);
void glColorMask(GLboolean red, GLboolean green, GLboolean blue,
GLboolean alpha);
void glDepthMask(GLboolean flag);
void glStencilMask(GLuint mask);

此函数设置掩码来控制对指定缓存的写入。由 glIndexMask()设置的掩码只能用于颜色索引模式中。在掩码中出现 1 处，颜色索引模式中相应的位被写入；在 0 出现的位，则不写。类似地，glColorMask()仅影响在 RGBA 模式下的绘图。red，green，blue，及 alpha 值控制相应的分量是否已写入（GL_TRUE 表示已写入）。如果 glDepthMask()中 flag 为 GL_TRUE，则深度缓存被激活用于写操作；否则取消。glStencilMask()的掩码以同样的方式应用于模板数据，这和 glIndexMask()的掩码用于颜色索引数据一样。所有 GLboolean 掩码的缺省值均为 GL_TRUE，两个 GLuint 掩码的缺省值均为 1。

在颜色索引模式下的颜色屏蔽操作中有许多诀窍。例如，可以将索引中的每一位当作不同的层，通过对颜色表进行适当的设置来设定任意层之间的相互作用。程序员还可以通过建立覆盖和底衬来做所谓的颜色图动画。使用颜色屏蔽的范例请参见第 14 章。RGBA 模式中的屏蔽较少使用，但可以用来将独立的图像文件装载到 red, green 和 blue 位平面中。

在带有深度缓存的三维混合一节中，我们已经看到了取消深度缓存的一种用法。如果希望将共同的背景用于一系列帧图像，并且希望增加某些被部分背景遮挡的特征，取消写入深度缓存也是非常有用的。例如，假如背景是一片森林，就希望反复绘制有相同树木和运动物体的多帧画面。根据记录在深度缓存中的深度值绘出树木之后，树木的图像就被保存起来，再以取消写入的深度缓存绘制新的项目。只要新的项目不相互重叠，图像就是正确的。为了绘制下一帧，则需要复制树木的图像，如此继续下去。在该过程中不需要恢复深度缓存中的值。当背景复杂的程度达到了将图像重新复制到颜色缓存中比根据几何学反复计算来得更快时，这个办法就特别有用。

对模板缓存的屏蔽允许程序员使用一个多位的模板缓存来保存多个一位的模板缓存。

注意：由 glStencilMask()指定的掩码用于控制哪个位平面可写。该掩码与作为 glStencilFunc()的第三个参数所指定的掩码无关，该参数指定在模板函数中该考虑哪个位平面。

11.2 片原的测试与操作

在屏幕上绘制几何体、文本及图像时，OpenGL 要完成一些计算，以便实现旋转、平移、缩放，确定光照，将物体投影到透视图中，判断窗口中哪些像素会受影响，以及确定这些像素应该以何种颜色绘制。本书前面的章节已经给出了关于如何控制这些操作的一些信息。在 OpenGL 确定了应该生成一个片原以及按何种颜色绘制之后，仍然需要几个处理过程来控制如何作为一个像素将该片原绘制到帧缓存中，以及是否需要这样做。如果该片原超出了窗口，或者比帧缓存中的像素距视点的距离更远，则该片原就不必画出。在另一个阶段中，片原的颜色要与已经在帧缓存中的像素颜色混合到一起。

本节描述了在放入帧缓存之前片原所必须通过的完整测试集合，并且描述了在片原写入时可能进行的最后操作。测试和操作按下列次序进行；如果在前面的测试中片原被删除，则不再进行后续的测试或操作。

1）剪取测试。

2）alpha 测试。

3）模板测试。

4）深度测试。

5）混合。

6）抖动。

7）逻辑操作。

下面各节对这些测试和操作进行了详细描述。

11.2.1 剪取测试

利用 glScissor()命令，可以定义窗口中一个矩形区域，并将作图限制在其中。如果片原落在该矩形区，则剪取测试通过。

---

void glScissor(GLint x, GLint y, GLsizei width, GLsizei height);

此函数设置剪取矩形的位置和尺寸。这些参数定义了该矩形的左下角(x, y)，宽度和高度。落在矩形区域中的像素将通过剪取测试。利用参数为 GL_SCISSOR 的函数 glEnable() 和 glDisable()激活和取消剪取测试。缺省情况下，矩形与窗口的大小相同，剪取处于取消状态。

剪取测试只是使用屏幕矩形区域进行的模板测试的翻版。在系统模板较慢时——也许因为模板是以软件方式执行的缘故——将很容易地建立剪取测试的封闭的硬件快速实现。

剪取的一种高级用法是用于非线性投影。首先，将窗口分成由子区域构成的矩形网格，指定视区和剪取参数使得同一时刻只对一个区域进行渲染。然后利用不同的投影矩阵，将整个场景投影到各个区域。

在函数 glIsEnabled()中使用 GL_SCISSOR_TEST，在函数 glGetIntegerv()中使用 GL_SCISSOR_BOX，可以确定是否剪取被激活，并获得定义剪取矩形的值。

11.2.2 alpha 测试

在 RGBA 模式下，将根据 alpha 测试中的 akalpha 值来确定接收还是拒绝一个片原。如果 alpha 测试被激活，该测试将输入的 alpha 值与参考值进行比较。根据比较结果，决定是否接收该片原。参考值和比较函数均由函数 glAlphaFunc()设置。带有参数 GL_ALPHA_TEST 的函数 glIsEnabled()用于确定 alpha 测试是否激活；参数分别为 GL_ALPHA_TEST_FUNC 和 GL_ALPHA_TEST_REF 的函数 glGetIntegerv()则用于获取比较函数和参考值。

---

void glAlphaFunc(GLenum func, GLclampf ref);

此函数设置 alpha 测试的参考值和比较函数。参考值 ref 取值截取在 0.0 和 1.0 之间。参数 func 的可能值及其含义见表 11-2。

表 11-2　与 glAlphaFunc()一起使用的参数值

参数	含义
GL_NEVER	总不接受该片原
GL_ALWAYS	总是接受该片原
GL_LESS	若片原 alpha < 参考 alpha，接受该片原
GL_LEQUAL	若片原 alpha ≤ 参考 alpha，接受该片原
GL_EQUAL	若片原 alpha=参考 alpha，接受该片原
GL_GEQUAL	若片原 alpha≥参考 alpha，接受该片原
GL_GREATER	若片原 alpha>参考 alpha，接受该片原
GL_NOTEQUAL	若片原 alpha ≠ 参考 alpha，接受该片原

参数 GL_ALPHA_TEST 传递给函数 glEnable 时激活 alpha 测试，传给 glDisable()时取消 alpha 测试。缺省情况下，参考值为 0.0，比较函数为 GL_ALWAYS，alpha 测试被取消。

alpha 测试的应用之一就是用来实现透明算法。对整个场景做 2 次渲染，第一次仅接收 alhpa=1.0 的片原，第二次接受 alhpa≠1.0 的片原。在两个过程中都打开深度缓存，但在第二个过程中，要使写操作处在取消状态。

另一个应用是用纹理图制作贴花，其中部分贴花是视线能够穿透的。在希望看透贴花的地方，将 alpha 值设置为 0.0，否则设置成 1.0。将参考值设置成 0.5（或者 0.0 和 1.0 之间的任何数），将比较函数设置为 GL_GREATER。这样，贴花就有一部分是可以看透的，并且深度缓存中的值不受影响。这种技术称之为"贴花法"，在混合的简单使用中将进行详细描述。

11.2.3　模板测试

模板测试只在有模板缓存的情况下发生（如果没有模板缓存，模板测试总能通过）。模板化适用于这样的测试，这种测试对存储于模板缓存中的像素值与参考值进行比较。根据比较结果，对模板缓存中的值进行修改。可以利用 glStencilFunc()及 glStencilOp()命令对需要用到的特定比较函数、参考值、以及所执行的修正进行选择。

---

void glStencilFunc(GLenum func, GLint ref, GLuint mask);

此函数为模板测试设置比较函数（func）,参考值(ref)，以及掩码(mask)。利用比较函数

可以对参考值和模板缓存中的值进行比较，而比较仅适用于掩码的相应位为 1 的位。比较函数可以是 GL_NEVER，GL_ALWAYS，GL_LESS，GL_LEQUAL，GL_EQUAL，GL_GEQUAL，GL_GREATER 或 GL_NOTEQUAL。例如，在函数为 GL_LESS 的情况下，如果 ref 的值小于模板缓存中的值，则片原通过测试。如果模板缓存包含 s 个位平面，在进行比较之前，掩码的低位中的 s 位与模板缓存中的值及参考值进行按位"与"操作。屏蔽值均按非负值处理。通过把参数 GL_STENCIL_TEST 传递给 glEnable()和 glDisable()来激活或取消模板测试。缺省情况下，func 为 GL_ALWAYS，ref=0.0, mask=1，模板化处于取消状态。

void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);

此函数说明当片原通过或未通过模板测试时，如何对模板缓存中的数据进行修正。三个函数 fail, zfail，zpass 可以为 GL_KEEP，GL_ZERO，GL_REPLACE，GL_INCR，GL_DECR，或 GL_INVERT。它们对应于：保持当前值、以 0 替换、以参考值替换、增加该值、减小该值、以及对其按位取反等操作。递增和递减函数被限制在 0 和最大无符号整数值之间（如果模板缓存包含 s 位，则为 2^s-1）。如果片原未通过模板测试，则应用 fail 函数；如果通过此测试，但深度测试失败，则应用 zfail；如果深度测试也通过，或者没有深度测试，则应用 zpass。缺省情况下，三个模板操作都是 GL_KEEP。

1. 模板查询

利用查询函数 glGetIntegerv()和表 11-3 中列出的值之一，可以获得 6 个与模板相关的参数的值。通过带有参数 GL_STENCIL_TEST 的函数 glIsEnabled()调用，可以确定是否激活了模板测试。

表 11-3 用于模板测试的查询值

查询值	含义
GL_STENCIL_FUNC	模板函数
GL_STENCIL_REF	模板参考值
GL_STENCIL_VALUE_MASK	模板掩码
GL_STENCIL_FAIL	模板操作失败
GL_STENCIL_PASS_DEPTH_FAIL	模板通过而深度缓存操作失败
GL_STENCIL_PASS_DEPTH_PASS	模板通过深度缓存也通过操作

2. 模板实例

模板测试较典型的应用就是屏蔽掉屏幕的不规则区域，以避免在该区域作图。要做到这一点，应以 0 填充模板掩码，然后以 1 在模板缓存中绘制出期望的形状。注意不能直接在模板缓存中作图，需要先在颜色缓存中作图，然后为 zpass 函数选择合适的值（例如

GL_REPLACE），来实现此结果。不管在什么时候进行作图，一定要将一些值写入模板缓存中（本例中为参考值）。为了避免模板缓存作图影响颜色缓存中的内容，要将颜色掩码设置为 0（或 GL_FALSE），同时还要取消向深度缓存中写入。

定义好模板缓存后，将参考值设置为 1，并且设定比较函数，使得只要参考值与模板平面值相等，该比较函数就会使片原得以通过模板测试。在作图过程中，不要修改模板平面的内容。

例 11-1 演示了如何以这种方式使用模板测试。该程序绘制了两个圆环，在场景的中间有钻石形的切除部分。在钻石形的模板掩码内，绘制了一个球。在本例中，只有在窗口重画时才在模板缓存中作图，因此在模板掩码创建之后要清除颜色缓存。

例 11-1　使用模板测试

```
#include "windows.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

#define YELLOWMAT   1
#define BLUEMAT 2

void CALLBACK myinit (void)
{
   GLfloat yellow_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
   GLfloat yellow_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat blue_diffuse[] = { 0.1, 0.1, 0.7, 1.0 };
   GLfloat blue_specular[] = { 0.1, 1.0, 1.0, 1.0 };
   GLfloat position_one[] = { 1.0, 1.0, 1.0, 0.0 };
   glNewList(YELLOWMAT, GL_COMPILE);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, yellow_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, yellow_specular);
   glMaterialf(GL_FRONT, GL_SHININESS, 64.0);
   glEndList();
   glNewList(BLUEMAT, GL_COMPILE);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, blue_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, blue_specular);
   glMaterialf(GL_FRONT, GL_SHININESS, 45.0);
```

```
    glEndList();
    glLightfv(GL_LIGHT0, GL_POSITION, position_one);
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHTING);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
    glClearStencil(0x0);
    glEnable(GL_STENCIL_TEST);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
            // 在模板为 1 处绘制蓝色球
    glStencilFunc (GL_EQUAL, 0x1, 0x1);
    glCallList (BLUEMAT);
    auxSolidSphere (0.5);
              // 在模板不是 1 处绘制圆环
    glStencilFunc (GL_NOTEQUAL, 0x1, 0x1);
    glStencilOp (GL_KEEP, GL_KEEP, GL_KEEP);
    glPushMatrix();
    glRotatef (45.0, 0.0, 0.0, 1.0);
    glRotatef (45.0, 0.0, 1.0, 0.0);
    glCallList (YELLOWMAT);
    auxSolidTorus (0.275, 0.85);
    glPushMatrix();
    glRotatef (90.0, 1.0, 0.0, 0.0);
    auxSolidTorus (0.275, 0.85);
    glPopMatrix();
    glPopMatrix();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
   glViewport(0, 0, w, h);
```

```
    glClear(GL_STENCIL_BUFFER_BIT); /* 创建一个钻石形的模板区域*/
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-3.0, 3.0, -3.0, 3.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glStencilFunc (GL_ALWAYS, 0x1, 0x1);
    glStencilOp (GL_REPLACE, GL_REPLACE, GL_REPLACE);
    glBegin(GL_QUADS);
    glVertex3f (-1.0, 0.0, 0.0);
    glVertex3f (0.0, 1.0, 0.0);
    glVertex3f (1.0, 0.0, 0.0);
    glVertex3f (0.0, -1.0, 0.0);
    glEnd();
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (GLfloat) w/(GLfloat) h, 3.0, 7.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -5.0);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH | AUX_STENCIL);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 11-1");
    myinit ();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return 0;
}
```

例 11-1 的运行结果如图 11-2 所示。

下面的例子说明了模板测试的其他用法。更多的讨论请参见第 14 章。

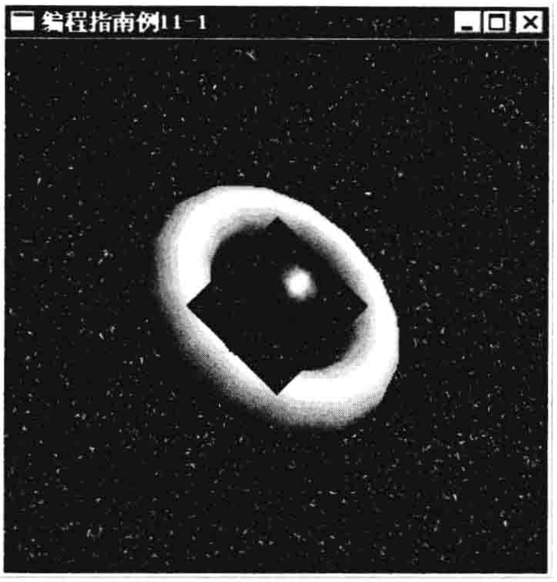


图 11-2 例 11-1 的运行结果

- 封顶——假定要绘制一个由一些多边形组成的封闭凸出物体（或几个，只要相互不相交即可），用一个裁剪平面将其部分切除。我们假定平面与物体相交，并希望用某个固定颜色的表面盖住物体，使之内部看不见。要实现这一点，要先将模板缓存清 0，模板化后开始作图，将模板比较函数设置为总可以接收片原。在接收片原时，对模板平面中的值取反。在所有物体绘制完成之后，不需要盖住的屏幕区域在模板平面中均为 0，而需要盖住的区域不为 0。重新设置模板函数，以便只绘制模板值不为 0 处，并在整个屏幕上以封顶颜色绘制一个大的多边形。
- 与半透明多边形的重叠——假定有一个由部分交叠的多边形组成的半透明表面。如果简单地使用 alpha 混合，该表面下面的物体，某些部分将被多个透明表面覆盖，这在视觉上是不对的。利用模板平面可以确保每一个片原顶多只被透明表面的一个部分所覆盖。这一点可以通过将模板平面清 0、只在模板平面为 0 时绘图、以及作图时增加模板平面的值来实现。
- 点划法——假定希望以点划模式绘制图像。将点划模式写入模板缓存，然后根据模板缓存中的内容作图，就可以实现以点划法绘制的图像。在原始的点划模式画完之后，绘制图像时，并不改变模板缓存，因此物体就可按照模板缓存中的模式进行点划。

11.2.4 深度测试

对于屏幕上每个像素，深度缓存时刻追踪视点与占据该像素的物体之间的距离。那么，如果通过了深度测试，输入的深度值就将取代深度缓存中的相应值。

深度缓存通常用于消除隐藏表面。如果有一种新的候选颜色用于该像素的显示，仅在对应的物体比以前更近时，该像素才被画出。通过这种方法，在对整个场景进行渲染之后，只

有未被其他物体遮盖的物体才被画出。在初始化的过程中，深度缓存通常是以距离视点尽可能远的值来填充的，因此任何物体的深度都比初始状态更近。如果希望使用深度缓存，只需调用参数为 GL_DEPTH_TEST 的函数 glEnable()将其激活。记住，在重画每一帧之前，要清除深度缓存（参见清空缓存）。可以用 glDepthFunc()函数为深度测试选择不同的比较函数。

---

void glDepthFunc(GLenum func);

此命令为深度测试设置比较函数。func 的值必须为 GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER 或 GL_NOTEQUAL。如果 z 值与深度缓存中的值有确定的关系，则输入片原通过深度测试。此关系缺省为 GL_LESS，意味着如果 z 值小于深度缓存中的 z 值，则片原通过测试。在这种情况下，z 值表示物体到视点的距离，值越小，表示对应的物体距视点越近。

11.2.5　混合，抖动及逻辑操作

一旦输入片原通过了上述所有测试，它就可以与颜色缓存中的当前内容按某种方式合并起来。最简单的方法，也是缺省操作，就是将当前值覆盖掉。当然，如果所处的是 RGBA 模式，并且希望片原是半透明或消除了锯齿现象的，程序员可以将该片原值与缓存中的值作一平均（即混合）。对于可用颜色较少的系统，也可以牺牲分辨率为代价，通过颜色值的抖动增加可用颜色数量。最后，在颜色索引模式中，可以利用任意的按位逻辑操作将输入片原与已写入的像素进行合成。

1. 混合

混合将输入片原的 R、G、B、A 值与存储在当前位置的像素的对应值结合起来。可以使用不同的混合操作。混合操作是否发生取决于输入 alpha 值与存储在像素中的 alpha 值（如果有）。

2. 抖动

对于颜色位平面较少的系统，可以通过在图像中抖动颜色，以空间分辨率为代价增加颜色分辨率。尽管报纸只有黑白两种颜色，但可以用黑白点阵的结合来表现灰度阴影效果，从而可以显示黑白照片。将报纸的无灰度照片图像和原来的照片比较一下，可以明显看出空间分辨率受到的损失。因为即使以最低品质的颜色显示也至少需要几种不同的 RGB 颜色值，所以要增加可用颜色范围，空间分辨率将受到一些影响。

抖动操作是和硬件相关的，OpenGL 允许程序员所做的操作就只有开或者关。实际上，如果机器的分辨率已经相当高，激活抖动操作根本就没有任何意义。要激活或取消抖动，可以用 GL_DITHER 作为参数调用函数 glEnable()和 glDisable()。缺省情况下，抖动是激活的。

抖动操作在 RGBA 模式和颜色索引模式中都有应用：在某些与硬件相关方法中，颜色或颜色索引的改变概率是相近的。例如，在颜色索引模式中，如果抖动已经激活，绘制的颜

色索引为 4.4，那么 60%的像素以索引 4 绘制，而另 40%的像素以索引 5 绘制。在 RGBA 模式中，抖动对各个分量是单独进行的；要在颜色索引模式中利用抖动，通常要在颜色图中分配合适的颜色梯度，否则，将产生奇怪的结果。

在 RGBA 模式下，抖动是结果值写入颜色缓存前的最后一步。在颜色索引模式下，则还可以执行下面描述的任何一种逻辑操作。

3. 逻辑操作

在颜色索引模式下，颜色索引将以整数或位模式来解释。对于明暗效果及抖动，理解为整数通常是最好的，但是对于在不同层的合成图像的绘制来说，位模式的解释最有意义。逻辑操作，例如“OR”、“XOR”等，通常用于输入片原值或（和）颜色缓存中的当前值之间的操作。

这些片原操作在某些计算机上尤为有用，这些机器上的基本图像操作就是将数据矩形从窗口的一个位置复制到另一个位置：或者从窗口到内存，或者从内存到窗口。通常，复制并非将数据直接写入内存，而是允许程序员对输入数据进行任意的逻辑操作，然后以操作结果覆盖现存的数据。由于这种处理在硬件上实现起来相当简单，因此许多机器都用到这种操作。作为使用逻辑操作的一个例子，“XOR”是一种可取消的作图方式，只要再对同一图像进行“XOR”作图，则原始图像就可得到恢复。

利用 glLogicOp()可在 16 种逻辑操作中进行选择，利用参数 GL_LOGIC_OP 调用函数 glEnable() 和 glDisable()可以激活或取消逻辑操作。

---

void glLogicOp(GLenum opcode);

此函数在颜色索引模式下，给定输入（源）片原和存储在颜色缓存中的当前像素（目标），选择要执行的逻辑操作。表 11-4 列出了参数 opcode 的可能值及其操作。缺省值为 GL_COPY。

表 11-4 16 种逻辑操作

参数	操作	参数	操作
GL_CLEAR	0	GL_AND	
GL_COPY	s	GL_OR	
GL_NOOP	d	GL_NAND	
GL_SET	1	GL_NOR	
GL_COPY_INVERTED		GL_XOR	
GL_INVERT		GL_EQUIV	
GL_AND_REVERSE		GL_AND_INVERTED	
GL_OR_REVERSE		GL_OR_INVERTED	

11.3　累积缓存

累积缓存用于场景锯齿的消除，运动模糊度，照相景深的模拟，以及计算多光源下的平滑阴影等操作。

OpenGL 的图形操作并不直接写入累积缓存。通常，一系列的图像是在标准的颜色缓存中生成的，然后，将它们一次一帧地存入累积缓存中。当累加完成之后，再将结果复制回颜色缓存中以便观察。为了减少舍入误差，累积缓存比标准颜色缓存有更高的精度，当然，渲染一个场景需要用更长的时间，但质量更高。在应用程序中，可以在渲染质量和渲染时间之间作出一个折中。

可以像摄影师对胶片进行多次曝光那样使用累积缓存。通常，摄影师以同一个胶片拍摄同一场景的多幅照片，实现多次曝光。如果场景中有移动的物体，那么该物体看起来就是模糊的。比起摄影师用照相机所作的事情，计算机还可以对图像进行更多的处理。例如，计算机可以很高超地控制视点，而摄影师却不能随心所欲地抖动照相机。

关于如何清除累积缓存及如何使用 glAccum()控制累积缓存的详细信息，请参见清除缓存一节。

---

void glAccum(GLenum op, GLfloat value);

控制累积缓存。参数 op 用于选择操作，value 是该操作中用到的数值。可能的操作是 GL_ACCUM, GL_LOAD, GL_RETURN, GL_ADD, 及 GL_MULT:

- GL_ACCUM 从当前选定的缓存中读取像素，该缓存为了用 glReadBuffer()进行读取而选定（参见第 8 章），用 value 乘上 R、G、B、A 值，然后将结果加到累积缓存中。
- GL_LOAD 操作与上述类似，但是用结果值替换掉累积缓存中的值，而不是与之相加。
- GL_RETURN 从累积缓存中取值，以 value 乘以该值，然后将该结果放入为写操作而激活的颜色缓存中。
- GL_ADD 和 GL_MULT 以 value 简单地与累积缓存中的像素值相加或相乘。然后将其返回到累积缓存中。对于 GL_MULT，value 截取在[-1.0,1.0]之间。对 GL_ADD 则不进行截取。

11.3.1　场景反走样

为了对场景进行反走样，首先要清空累积缓存，将前缓存激活用于读写操作。然后，对在稍微不同的位置绘制的图像代码做几次（n 次）循环，以函数：

```
glAccum(GL_ACCUM, 1.0/n);
```

累加数据；最后，调用函数：

```
glAccum(GL_RETURN, 1.0);
```

注意：如果在第一次循环中就使用了 GL_LOAD，并且忽略对累积缓存的清空，则这种方法是十分快捷的。在本章后面的逻辑操作一节中，讨论了可能出现的抖动值。在这段代码中，在最终绘制图像之前，对图像进行 n 次绘制。如果避免用户看见中间图像，可将其绘制到颜色缓存中而不显示出来。对其进行累加，然后利用 GL_RETURN 调用将其画入显示的缓存中（或是切换到前面的后缓存中）。

另外，还可以提供一个用户界面来显示随着图像的累加而逐渐改进的图像，同时允许用户在图像已经满足要求时终止这个过程。为了实现这一点，在绘制连续图像的过程中，每次累加后利用 16.0/1.0, 16.0/2.0, 16.0/3.0, ...作为第二个参数，以参数 GL_RETURN 调用函数 glAccum()。利用这种技术在完成一次累加后，显示最终图像的 1/16，第二次显示 2/16，如此继续。在 GL_RETURN 之后，这段代码应该检查是否有用户中断，因为在每次循环之后都要复制结果图像，这个界面运行起来显得有些慢。

要确定 n 的值为多少，需要在速度和质量之间做一个权衡：绘制场景的时间越长，获得最后图像需要的时间就越长。绘制场景的次数越多，物体越平滑，可以一直平滑到累积缓存的分辨率的最大限度。彩图 22 和彩图 23 显示了场景进行反走样后质量的改进。

例 11-2 为抖动操作定义了两个子函数：ccPerspective() 和 accFrustum()。函数 ccPerspective()用于替代 gluPerspective()，两个函数的前 4 个参数都是相同的。要抖动视图体（用于场景反走样），应将 x 和 y 值作为 accPerspective()的第 5 和 6 个参数。同时，将 0.0 作为 accPerspective()的第 7 和第 8 个参数，而以非 0 值作为第 9 个参数。后 3 个参数用于景深效果，这一内容将在后面描述。

例 11-2　用于抖动视图体的函数：accpersp.c

```
void CALLBACK accFrustum(GLdouble left,  GLdouble right, GLdouble bottom,
                GLdouble top,   GLdouble near,  GLdouble far,
                GLdouble pixdx, GLdouble pixdy, GLdouble eyedx,
                GLdouble eyedy, GLdouble focus)
{
GLdouble xwsize, ywsize;
GLdouble dx, dy;
GLint viewport[4];
glGetIntegerv (GL_VIEWPORT, viewport);
xwsize = right - left;
ywsize = top - bottom;
dx = -(pixdx*xwsize/(GLdouble) viewport[2] +  eyedx*near/focus);
dy = -(pixdy*ywsize/(GLdouble) viewport[3] +   eyedy*near/focus);
```

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum (left + dx, right + dx, bottom + dy, top + dy, near, far);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef (-eyedx, -eyedy, 0.0);
}

void CALLBACK accPerspective(GLdouble fovy, GLdouble aspect, GLdouble near,
                GLdouble far,  GLdouble pixdx,  GLdouble pixdy,
                GLdouble eyedx, GLdouble eyedy,  GLdouble focus)
{
   GLdouble fov2,left,right,bottom,top;
   fov2 = ((fovy*PI_) / 180.0) / 2.0;
   top = near / (fcos(fov2) / fsin(fov2));
   bottom = -top;
   right = top * aspect;
   left = -right;
   accFrustum (left, right, bottom, top, near, far,
           pixdx, pixdy, eyedx, eyedy, focus);
}
```

例 11-3 利用上面给出的两个子程序来执行对场景反走样。

例 11-3　场景反走样：accpersp.c

```
#include <windows.h>

#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL/glaux.h>
#include "jitter.h"

void myinit(void);
```

```
void accFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
     GLdouble znear, GLdouble zfar, GLdouble pixdx, GLdouble pixdy,
     GLdouble eyedx, GLdouble eyedy, GLdouble focus);

void accPerspective(GLdouble fovy, GLdouble aspect,
     GLdouble znear, GLdouble zfar, GLdouble pixdx, GLdouble pixdy,
     GLdouble eyedx, GLdouble eyedy, GLdouble focus);

void displayObjects(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);


#define PI_ 3.14159265358979323846

void accFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
    GLdouble znear, GLdouble zfar, GLdouble pixdx, GLdouble pixdy,
    GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
    GLdouble xwsize, ywsize;
    GLdouble dx, dy;
    GLint viewport[4];

    glGetIntegerv (GL_VIEWPORT, viewport);

    xwsize = right - left;
    ywsize = top - bottom;

    dx = -(pixdx*xwsize/(GLdouble) viewport[2] + eyedx*znear/focus);
    dy = -(pixdy*ywsize/(GLdouble) viewport[3] + eyedy*znear/focus);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum (left + dx, right + dx, bottom + dy, top + dy, znear, zfar);
    glMatrixMode(GL_MODELVIEW);
```

```
    glLoadIdentity();
    glTranslatef (-eyedx, -eyedy, 0.0);
}

void accPerspective(GLdouble fovy, GLdouble aspect,
    GLdouble znear, GLdouble zfar, GLdouble pixdx, GLdouble pixdy,
    GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
    GLdouble fov2,left,right,bottom,top;

    fov2 = ((fovy*PI_) / 180.0) / 2.0;

    top = znear / (cos(fov2) / sin(fov2));
    bottom = -top;

    right = top * aspect;
    left = -right;

    accFrustum (left, right, bottom, top, znear, zfar,
    pixdx, pixdy, eyedx, eyedy, focus);
}

/*  初始化光照和其他值。
 */
void myinit(void)
{
    GLfloat mat_ambient[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 0.0, 0.0, 10.0, 1.0 };
    GLfloat lm_ambient[] = { 0.2, 0.2, 0.2, 1.0 };

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 50.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

```
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lm_ambient);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
    glShadeModel (GL_FLAT);

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClearAccum(0.0, 0.0, 0.0, 0.0);
}

void displayObjects(void)
{
    GLfloat torus_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
    GLfloat cube_diffuse[] = { 0.0, 0.7, 0.7, 1.0 };
    GLfloat sphere_diffuse[] = { 0.7, 0.0, 0.7, 1.0 };
    GLfloat octa_diffuse[] = { 0.7, 0.4, 0.4, 1.0 };

    glPushMatrix ();
    glTranslatef (0.0, 0.0, -5.0);
    glRotatef (30.0, 1.0, 0.0, 0.0);

    glPushMatrix ();
    glTranslatef (-0.80, 0.35, 0.0);
    glRotatef (100.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, torus_diffuse);
    auxSolidTorus (0.275, 0.85);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (-0.75, -0.50, 0.0);
    glRotatef (45.0, 0.0, 0.0, 1.0);
    glRotatef (45.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, cube_diffuse);
```

```
    auxSolidCube (1.5);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (0.75, 0.60, 0.0);
    glRotatef (30.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, sphere_diffuse);
    auxSolidSphere (1.0);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (0.70, -0.90, 0.25);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, octa_diffuse);
    auxSolidOctahedron (1.0);
    glPopMatrix ();

    glPopMatrix ();
}

#define ACSIZE  8

void CALLBACK display(void)
{
    GLint viewport[4];
    int jitter;

    glGetIntegerv (GL_VIEWPORT, viewport);

    if(!viewport[3]) return;

    glClear(GL_ACCUM_BUFFER_BIT);
    for (jitter = 0; jitter < ACSIZE; jitter++) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        accPerspective (50.0,
            (GLdouble) viewport[2]/(GLdouble) viewport[3],
```

```
            1.0, 15.0, j8[jitter].x, j8[jitter].y,
            0.0, 0.0, 1.0);
        displayObjects ();
        glAccum(GL_ACCUM, 1.0/ACSIZE);
        glFlush();
        auxSwapBuffers();
    }
    glAccum (GL_RETURN, 1.0);
    glFlush();
    auxSwapBuffers();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_DOUBLE | AUX_RGB
            | AUX_ACCUM | AUX_DEPTH16);
    auxInitPosition (0, 0, 250, 250);
    auxInitWindow ("编程指南例 11-3");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

例 11-3 的运行结果如图 11-3 所示。

其实不必利用透视投影来执行场景的反走样。通过利用函数 glTranslate*()来抖动场景，简单地以正交投影即可实现场景反走样。注意，glTranslate*()是在全局坐标中操作的，而我们希望场景的运动小于屏幕坐标下的一个像素。因此，必须利用全局坐标中宽度和高度值除以视区尺寸来计算抖动平移值，对全局坐标映射进行变换。将抖动数量与全局坐标值相乘，以便确定为了获得少于一个像素的抖动，场景的运动量应该为多少。例 11-4 给出了高度和宽度的全局坐标为 4.5 时的子函数 display()和 myReshape()。

图 11-3　例 11-3 的运行结果

例 11-4　以正交投影进行抖动

```
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include "jitter.h"

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

void myinit(void)
{
   GLfloat mat_ambient[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat light_position[] = { 0.0, 0.0, 10.0, 1.0 };
   GLfloat lm_ambient[] = { 0.2, 0.2, 0.2, 1.0 };

   glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialf(GL_FRONT, GL_SHININESS, 50.0);
```

```
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lm_ambient);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
    glShadeModel (GL_FLAT);

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClearAccum(0.0, 0.0, 0.0, 0.0);
}

void displayObjects(void)
{
    GLfloat torus_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
    GLfloat cube_diffuse[] = { 0.0, 0.7, 0.7, 1.0 };
    GLfloat sphere_diffuse[] = { 0.7, 0.0, 0.7, 1.0 };
    GLfloat octa_diffuse[] = { 0.7, 0.4, 0.4, 1.0 };

    glPushMatrix ();
    glRotatef (30.0, 1.0, 0.0, 0.0);

    glPushMatrix ();
    glTranslatef (-0.80, 0.35, 0.0);
    glRotatef (100.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, torus_diffuse);
    auxSolidTorus (0.275, 0.85);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (-0.75, -0.50, 0.0);
    glRotatef (45.0, 0.0, 0.0, 1.0);
    glRotatef (45.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, cube_diffuse);
```

```
    auxSolidCube (1.5);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (0.75, 0.60, 0.0);
    glRotatef (30.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, sphere_diffuse);
    auxSolidSphere (1.0);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (0.70, -0.90, 0.25);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, octa_diffuse);
    auxSolidOctahedron (1.0);
    glPopMatrix ();

    glPopMatrix ();
}

#define ACSIZE  8

void CALLBACK display(void)
{
    GLint viewport[4];
    int jitter;

    glGetIntegerv (GL_VIEWPORT, viewport);

    glClear(GL_ACCUM_BUFFER_BIT);
    for (jitter = 0; jitter < ACSIZE; jitter++) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glPushMatrix ();
/* 注意 4.5 为全局坐标下左右之间和上下之间的距离，下面的公式将小数
  的像素运动转换到全局坐标下 。
*/
```

```
        glTranslatef (j8[jitter].x*4.5/viewport[2],
            j8[jitter].y*4.5/viewport[3], 0.0);
        displayObjects ();
        glPopMatrix ();
        glAccum(GL_ACCUM, 1.0/ACSIZE);
        glFlush();
        auxSwapBuffers();
    }
    glAccum (GL_RETURN, 1.0);
    glFlush();
    auxSwapBuffers();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    if (!h) return;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho (-2.25, 2.25, -2.25*h/w, 2.25*h/w, -10.0, 10.0);
    else
    glOrtho (-2.25*w/h, 2.25*w/h, -2.25, 2.25, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_DOUBLE | AUX_RGB
            | AUX_ACCUM | AUX_DEPTH16);
    auxInitPosition (0, 0, 250, 250);
    auxInitWindow ("编程指南例 11-4");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
```

```
    return(0);
}
```

例 11-4 的运行结果如图 11-4 所示。

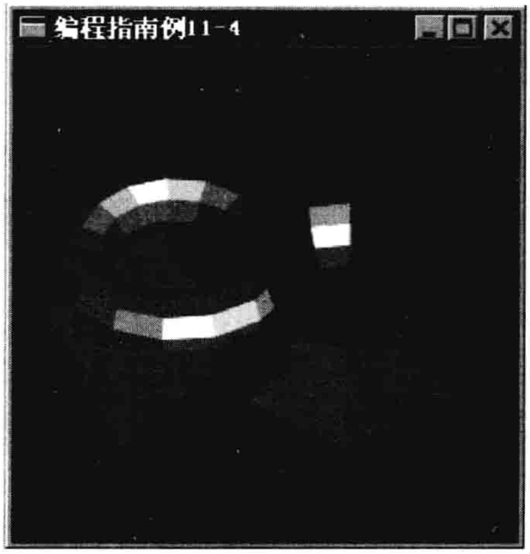


图 11-4 例 11-4 的运行结果

11.3.2 运动模糊度

可以利用类似的方法来模拟运动的模糊，如彩图 7 及图 11-5 所示。假定场景中存在静止和运动的物体，我们希望制作有一个小时间长度的运动模糊图像。累积缓存以同样的方法进行设置，但不是空间抖动图像。通过调用函数：

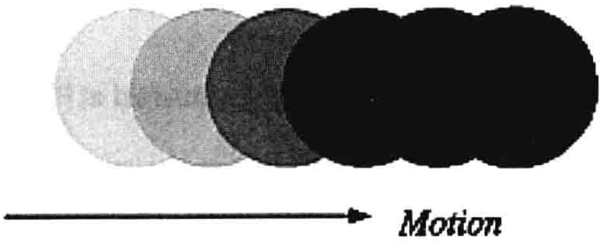


图 11-5 一个运动模糊的物体

```
glAccum (GL_MULT, decayFactor);
```

在场景被绘制到累积缓存中时，将整个场景连续地制作成调光器。其中，延迟因子是 0.0 到 1.0 之间的一个数。数值越小，物体看起来运动得越快。可以从物体当前位置移动全部场景，用函数：

```
glAccum (GL_RETURN, 1.0);
```

将先前位置的“水蒸汽似的痕迹”从累积缓存移到标准的颜色缓存中。

即使以不同的速度移动物体，或者使某些物体加速，图像看起来也是正确的。和前面的一样，所使用的抖动点越多，最后的图像质量越好。可以在空间及时间域上进行抖动，将运动模糊与消除锯齿现象结合起来。但图像质量越好，需要渲染的时间就越长。

11.3.3 景深

只有从距离胶片一定距离的单个平面上，照相机拍摄的照片才有正确的焦距，离该平面越远，焦距的误差越大。照相机的景深是处于正确焦平面的焦距范围，在该区域内物体超出焦距的量是非常小的。

在正常情况下，利用 OpenGL 绘制的任何事物都处在焦点上（除非显示器性能很差，这时候绘制的任何事物都在焦距之外）。累积缓存可用于近似在照片上看到的情况：在照片上，随着与正确焦平面的距离的增加，物体会越来越模糊。利用累积缓存并不是对照相机效果的精确模拟，但会使结果类似于照相机的拍摄。

为了获得这种效果，可以利用不同的参数值调用函数 glFrustum()，反复地绘制场景。选择参数使得视点在真正视点周围做轻微的变化，使每一次的视图截体共享一个焦平面上的矩形，如图 11-6 所示。渲染的结果应以通常的方法用累积缓存进行平均。

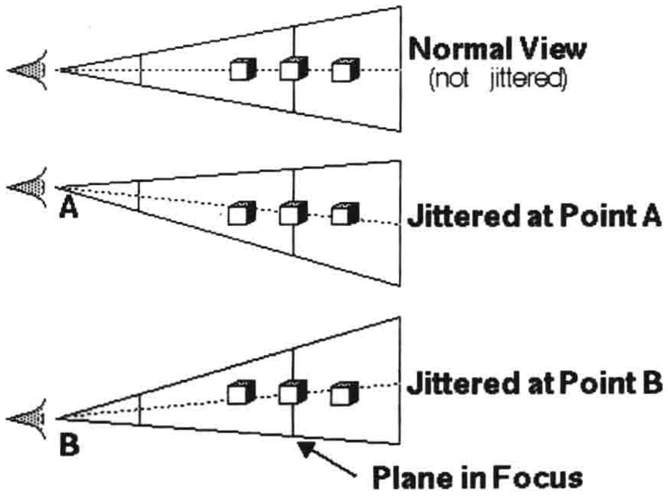


图 11-6 用于景深效果抖动的视图体

彩图 10 显示了利用景深效果绘制的 5 个茶壶的图像。金色的茶壶（左数第二个）在焦点处，其他的茶壶根据其距焦平面的距离逐渐变得模糊。绘制该图像的代码见例 11-5（假定子程序 accPerspective()及 accFrustum()在例 11-2 中定义）。场景被绘制了 8 次，每一个通过调用 accPerspective()对视图体做轻微抖动。在消除场景的锯齿现象这一过程中，第 5 和第 6 个参数分别在 x 和 y 方向抖动视图体。然而，对于景深效果而言，希望在将其保持在焦平面的同时抖动视图体。焦平面具有由函数 accPerspective()的第 9 个参数定义的值，在本例中，z=0。模糊量由一个常数与 x、y 方向的抖动值（accPerspective()函数的第 7 和第 8 个参数）

的乘积来确定。该常数是根据经验确定的（注意，在例 11-5 中，accPerspective()的第 5 和第 6 个参数均被置为 0.0，因此场景锯齿现象的消除操作是关闭的）。

例 11-5　创建景深效果

```
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL/glaux.h>
#include "jitter.h"

void myinit(void);
void accFrustum(GLdouble left, GLdouble right, GLdouble bottom,
   GLdouble top, GLdouble znear, GLdouble zfar, GLdouble pixdx,
   GLdouble pixdy, GLdouble eyedx, GLdouble eyedy, GLdouble focus);
void accPerspective(GLdouble fovy, GLdouble aspect,
   GLdouble znear, GLdouble zfar, GLdouble pixdx, GLdouble pixdy,
   GLdouble eyedx, GLdouble eyedy, GLdouble focus);
void renderTeapot (GLfloat x, GLfloat y, GLfloat z,
   GLfloat ambr, GLfloat ambg, GLfloat ambb,
   GLfloat difr, GLfloat difg, GLfloat difb,
   GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

#define PI_ 3.14159265358979323846

void accFrustum(GLdouble left, GLdouble right, GLdouble bottom,
   GLdouble top, GLdouble znear, GLdouble zfar, GLdouble pixdx,
   GLdouble pixdy, GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
   GLdouble xwsize, ywsize;
   GLdouble dx, dy;
   GLint viewport[4];
```

```
    glGetIntegerv (GL_VIEWPORT, viewport);

    xwsize = right - left;
    ywsize = top - bottom;

    dx = -(pixdx*xwsize/(GLdouble) viewport[2] + eyedx*znear/focus);
    dy = -(pixdy*ywsize/(GLdouble) viewport[3] + eyedy*znear/focus);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum (left + dx, right + dx, bottom + dy, top + dy, znear, zfar);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (-eyedx, -eyedy, 0.0);
}

void accPerspective(GLdouble fovy, GLdouble aspect,
    GLdouble znear, GLdouble zfar, GLdouble pixdx, GLdouble pixdy,
    GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
    GLdouble fov2,left,right,bottom,top;

    fov2 = ((fovy*PI_) / 180.0) / 2.0;

    top = znear / (cos(fov2) / sin(fov2));
    bottom = -top;

    right = top * aspect;
    left = -right;

    accFrustum (left, right, bottom, top, znear, zfar,
    pixdx, pixdy, eyedx, eyedy, focus);
}

void myinit(void)
```

```
{
    GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };

    GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
    GLfloat local_view[] = { 0.0 };

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
    glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, local_view);

    glFrontFace (GL_CW);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClearAccum(0.0, 0.0, 0.0, 0.0);
}

void renderTeapot (GLfloat x, GLfloat y, GLfloat z,
   GLfloat ambr, GLfloat ambg, GLfloat ambb,
   GLfloat difr, GLfloat difg, GLfloat difb,
```

```
    GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
{
    float mat[3];

    glPushMatrix();
    glTranslatef (x, y, z);
    mat[0] = ambr; mat[1] = ambg; mat[2] = ambb;
    glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
    mat[0] = difr; mat[1] = difg; mat[2] = difb;
    glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
    mat[0] = specr; mat[1] = specg; mat[2] = specb;
    glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
    glMaterialf (GL_FRONT, GL_SHININESS, shine*128.0);
    auxSolidTeapot(0.5);
    glPopMatrix();
}

void CALLBACK display(void)
{
    int jitter;
    GLint viewport[4];

    glGetIntegerv (GL_VIEWPORT, viewport);
    if (!viewport[3]) return;
    glClear(GL_ACCUM_BUFFER_BIT);

    for (jitter = 0; jitter < 8; jitter++) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    accPerspective (45.0,
        (GLdouble) viewport[2]/(GLdouble) viewport[3],
        1.0, 15.0, 0.0, 0.0,
        0.33*j8[jitter].x, 0.33*j8[jitter].y, 5.0);
    renderTeapot (-1.1, -0.5, -4.5, 0.1745, 0.01175, 0.01175,
        0.61424, 0.04136, 0.04136, 0.727811, 0.626959, 0.626959, 0.6);
    renderTeapot (-0.5, -0.5, -5.0, 0.24725, 0.1995, 0.0745,
```

```
        0.75164, 0.60648, 0.22648, 0.628281, 0.555802, 0.366065, 0.4);
    renderTeapot (0.2, -0.5, -5.5, 0.19225, 0.19225, 0.19225,
        0.50754, 0.50754, 0.50754, 0.508273, 0.508273, 0.508273, 0.4);
    renderTeapot (1.0, -0.5, -6.0, 0.0215, 0.1745, 0.0215,
        0.07568, 0.61424, 0.07568, 0.633, 0.727811, 0.633, 0.6);
    renderTeapot (1.8, -0.5, -6.5, 0.0, 0.1, 0.06, 0.0, 0.50980392,
        0.50980392, 0.50196078, 0.50196078, 0.50196078, .25);
    glAccum (GL_ACCUM, 0.125);
    glFlush();
    auxSwapBuffers();
    }

    glAccum (GL_RETURN, 1.0);
    glFlush();
    auxSwapBuffers();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_DOUBLE | AUX_RGB
            | AUX_ACCUM | AUX_DEPTH16);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow ("编程指南例 11-5");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

程序 10-5 的运行结果如图 11-7 所示。

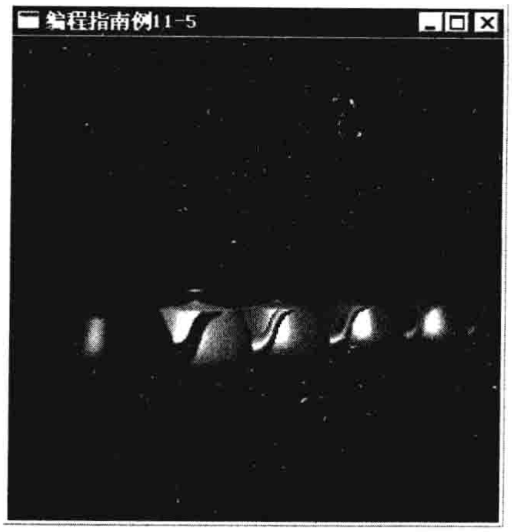


图 11-7　例 11-5 的运行结果

11.3.4　柔和阴影

要将多光源产生的柔和阴影累加起来，先一次打开一个光源并对阴影进行渲染，然后将它们累加到一起。这一过程可以与场景锯齿现象的消除和所进行的空间抖动同时进行。

11.3.5　抖动

如果需要进行 9 次或 16 次采样以消除图像的锯齿现象，有人可能认为最好是采用当前像素点的等空间网格。奇怪的是，事实并非如此——实际上常采用邻近的像素点。另外，可能需要进行均匀分配，或规格化分配等等。表 11-5 列出了对于某些选定的采样数的合理的抖动值的集合。表中大多数例子都是均匀分布采样。

表 11-5　采样抖动值

采样数	值
2	{0.25, 0.75}, {0.75, 0.25}
3	{0.5033922635, 0.8317967229}, {0.7806016275, 0.2504380877},{0.2261828938, 0.4131553612}
4	{0.375, 0.25}, {0.125, 0.75}, {0.875, 0.25}, {0.625, 0.75}
5	{0.5, 0.5}, {0.3, 0.1}, {0.7, 0.9}, {0.9, 0.3}, {0.1, 0.7}
6	{0.4646464646, 0.4646464646}, {0.1313131313, 0.7979797979},{0.5353535353, 0.8686868686}, {0.8686868686, 0.5353535353},{0.7979797979, 0.1313131313}, {0.2020202020, 0.2020202020}
8	{0.5625, 0.4375}, {0.0625, 0.9375}, {0.3125, 0.6875}, {0.6875, 0.8125},{0.8125, 0.1875}, {0.9375,0.5625}, {0.4375, 0.0625}, {0.1875, 0.3125}

续表

采样数	值
9	{0.5, 0.5}, {0.1666666666, 0.9444444444}, {0.5, 0.1666666666}, {0.5, 0.8333333333}, {0.1666666666, 0.2777777777}, {0.8333333333, 0.3888888888}, {0.1666666666, 0.6111111111}, {0.8333333333, 0.7222222222}, {0.8333333333, 0.0555555555}
12	{0.4166666666, 0.625}, {0.9166666666, 0.875}, {0.25, 0.375},{0.4166666666, 0.125},{0.75, 0.125}, {0.0833333333, 0.125}, {0.75, 0.625}, {0.25, 0.875}, {0.5833333333, 0.375}, {0.9166666666, 0.375}, {0.0833333333, 0.625}, { 0.583333333, 0.875}
16	{0.375, 0.4375}, {0.625, 0.0625}, {0.875, 0.1875}, {0.125, 0.0625}, {0.375, 0.6875}, {0.875, 0.4375}, {0.625, 0.5625}, {0.375, 0.9375}, {0.625, 0.3125}, {0.125, 0.5625}, {0.125, 0.8125}, {0.375, 0.1875}, {0.875, 0.9375}, {0.875, 0.6875}, {0.125, 0.3125}, {0.625, 0.8125}

第 12 章　求值程序和非均匀有理 B 样条（NURBS）

本章主题

图形硬件是用来绘制点、线段和多边形的（主要是指三角形和四边形）。平滑的曲线和曲面可以由大量的小线段、小多边形近似得到。对于许多常用的、有特殊属性的曲线曲面，OpenGL 中提供了更大的方便，通过较少量的参数（如一些控制点）便可以进行描述。显然，为一个曲面保存 16 个控制点所需的存储量远远小于保存 1000 个三角形及其各个顶点法向信息的需求。并且，再多的三角形也仅仅是对真实曲面的近似，而通过特定控制点则可以精确地描述真实曲面。

求值程序便是用控制点来描述曲线或曲面（或部分）上的点的工具。利用求值程序，相应的曲线曲面可以任何精度进行渲染。另外，曲面的法向量可以通过自动计算得到。以不同方式利用求值程序产生的点，可以绘制出以点、线框图、或者是填充（带有灯光和阴影）等方式表示的曲面。

程序员还可以利用求值程序描述任何多项式曲面、有理多项式样条曲面、或任何角度的曲面。这几乎包括了所有目前使用的样条和样条曲面，如 B 样条、非均匀有理 B 样条曲面、贝塞尔曲线和曲面、埃尔米特样条等。求值程序仅为曲线或曲面上的点提供了低级的描述，但是，它常常应用于实用库中，为程序员提供高级的交互接口程序。GLU 的非均匀有理 B 样条（NURBS）工具就是这样的一个高级接口程序，非均匀有理 B 样条程序封装了大量的复杂编码，但最终的渲染是通过求值程序完成的。

为了讲述的方便，在后面我们简称非均匀有理 B 样条为其英文缩写——NURBS。

本章包括以下几个主要部分：

- 概述讨论了为本章内容的展开讨论所作的假设和相关知识的说明。
- 求值程序解释了求值程序是如何工作的，以及怎样通过恰当的 OpenGL 命令对其进行控制。
- GLU NURBS 接口程序描述了创建 NURBS 曲面的实用库（GLU）子程序。

12.1　概述

求值程序的产生是基于贝塞尔（或伯恩斯坦）曲线的原理样条和曲面。在本节中将给出贝塞尔曲线的原理函数的定义公式。这里我们没有包括最基本原理和数学特性。若需要用求值程序绘制基于其他原理的曲线和曲面，必须知道如何将该原理转化为贝塞尔曲线原理。另

外，当利用求值程序渲染贝塞尔曲面或其部分时，需要确定划分的力度。这时要考虑高质量（较细的划分）图像与高速度的平衡。确定恰当的划分策略也是相当复杂的，这里不作讨论。

同样，我们没有对 NURBS 进行完整的讨论，但是本章中介绍了 GLU 的 NURBS 接口程序，并且提供了相应例程。在后面的叙述中，我们假定读者已经对 NURBS 的控制点、节的次序和修剪曲线等相关知识有所了解。

注意：由于一些术语并未形成统一的协议，本书采用的关于样条曲线和曲面的术语可能会与其他著作有所不同，通常，OpenGL 对概念内涵的限制常常更为严格。例如，OpenGL 的求值程序原理仅指贝塞尔原理。在其他场合下，求值程序可以基于任意的原理。

12.2　求值程序

贝塞尔曲线是具有一个变量的向量值函数：

$$C(u) = [X(u) \quad Y(u) \quad Z(u)]$$

其中 u 在某一范围内变化（如[0，1]）。贝塞尔曲面片原是有两个变量的向量值函数：

$$S(u,v) = [X(u,v) \quad Y(u,v) \quad Z(u,v)]$$

其中 u 和 v 都在一定范围内变化。这里的范围是三维的。曲线或纹理坐标的输出也可以是二维平面的，而 RGBA 信息的输出则可以是四维的，甚至于可以是表示灰度的一维输出。

对于每个 u（如果是曲面，则对于 u 和 v），公式 C(u)（或 S(u,v)）计算曲线（或曲面）上的一个点。使用求值程序时，首先需要定义这两个函数，然后激活求值程序，最后用 glEvalCoord1()或 glEvalCoord2()命令代替 glVertex()。这样一来，曲线或曲面的顶点便可以同其他的顶点一样形成点或线。同时，其他命令自动产生一系列顶点，形成均匀间隔为 u 的规则网格。一维和二维求值程序是类似的。考虑到讨论的方便，所以我们首先讨论一维的情况。

12.2.1　一维求值程序

这一节列出了一个利用一维求值程序绘制曲线的例子。然后，介绍了控制求值程序的命令和方程式。

1. 一维求值程序的例子：简单贝塞尔曲线

例 12-1 中的程序用四个控制点绘制一条三次贝塞尔曲线，如图 12-1 所示。

例 12-1　用四个控制点绘制贝塞尔曲线：bezcurve.c

```
#include "glos.h"
#include "windows.h"


#include <GL/gl.h>
#include <GL/glu.h>
```

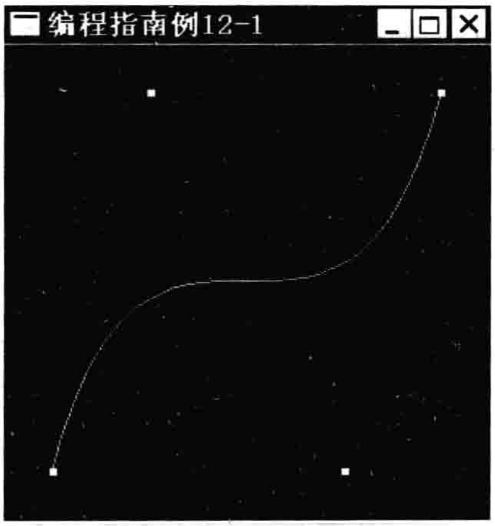


图 12-1　贝塞尔曲线

```
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLfloat ctrlpoints[4][3] = {
    { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
    {2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}};

void myinit(void)
{
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
    glEnable(GL_MAP1_VERTEX_3);
    glShadeModel(GL_FLAT);
}

void CALLBACK display(void)
{
    int i;

    glClear(GL_COLOR_BUFFER_BIT);
```

```
    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_LINE_STRIP);
    for (i = 0; i <= 30; i++)
        glEvalCoord1f((GLfloat) i/30.0);
    glEnd();
    /* 下列代码显示控制点。 */
    glPointSize(5.0);
    glColor3f(1.0, 1.0, 0.0);
    glBegin(GL_POINTS);
    for (i = 0; i < 4; i++)
        glVertex3fv(&ctrlpoints[i][0]);
    glEnd();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    if (!h) return;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
        5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
    else
    glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
        5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 300, 300);
```

```
    auxInitWindow ("编程指南例 12-1");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

三次贝塞尔曲线用四个控制点描述，在本例中，用数组 ctrlpoints[][]表示这四个控制点。此数组是 glMap1f()的参数。命令的所有参量如表 12-1 所示。

表 12-1 glmaplf()的参量

参量	功能
GL_MAP1_VERTEX_3	产生的三维顶点
0	参数 u 的最小值
1	参数 u 的最大值
3	控制点的数据到下一个控制点需要上升的浮点值数目
4	样条的阶数，等于次数+1，这里次数是 3（立方）
&ctrlpoints[0][0]	指向第一个控制点的数据

注意第二个和第三个参量控制曲线的参数化——当变量 u 从 0 变到 1，曲线从一个端点到另一个端点。调用 glEnable()为二维顶点激活了一维求值程序。

曲线在函数 display()中画出，display()在 glBegin()和 glEnd()之间调用。因为求值程序处于激活状态，glEvalCoord1f()的作用类似于 glVertex()，同时带有与输入参数 u 相关的曲线顶点坐标。

2. 一维求值程序的定义和求值

伯恩斯坦多项式的次数 n（或阶数 n+1）由下式给出：

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

如果 Pi 表示一组控制点（一维、二维、三维或四维），当 u 从 0 到 1 变化时，方程

$$C(u) = \sum_{i=0}^{n} B_i^n(u) P_i$$

代表贝塞尔曲线。代表同样的曲线，但允许 u 从 u1 变到 u2，而不是从 0 到 1，便可求值：

$$C\left(\frac{u-u_1}{u_2-u_1}\right)$$

命令 glMap1()定义了应用这些方程的一维求值程序。

void glMap1{fd}(GLenum target, TYPE u1, TYPE u2, GLint stride, GLint order, const TYPE *points);

此函数定义了一个一维求值程序。参数 target 指定了控制点所表述的内容（如表 12-2 所示）以及这些点需要提供多少数值。这些点可以来代表顶点、RGBA 颜色数据、法线向量或纹理坐标。例如，由 GL_MAP1_COLOR_4，求值程序产生在四维颜色空间中沿曲线的颜色数据。在调用之前，还需要用到表 12-1 中的参数值来激活求值程序。传递适当的值给函数 glEnable()或 glDisable()可以激活或取消求值程序。

接下来的两个参数 u1 和 u2 指出变量 u 的范围。变量 stride 是每块存储量中单精度或双精度数（随情况而定）的数目。也就是控制点的起始位置之间的偏移。

参数 order 是次数加 1，且与控制点的数目一致。参数 points 指向第一个控制点的第一个坐标。按照 glMap1*()的示例数据结构，可以这样表示点：

(GLfloat *)(&ctlpoints[0].x)

表 12-2　glMap1*()所用的控制点的类型

参数	含义
GL_MAP1_VERTEX_3	x, y, z 顶点坐标
GL_MAP1_VERTEX_4	x, y, z, w 顶点坐标
GL_MAP1_INDEX	颜色索引
GL_MAP1_COLOR_4	R, G, B, A
GL_MAP1_NORMAL	法向坐标
GL_MAP1_TEXTURE_COORD_1	s 纹理坐标
GL_MAP1_TEXTURE_COORD_2	s, t 纹理坐标
GL_MAP1_TEXTURE_COORD_3	s, t, r 纹理坐标
GL_MAP1_TEXTURE_COORD_4	s, t, r, q 纹理坐标

可以同时测试多个求值程序。假如同时定义和激活了求值程序 GL_MAP1_VERTEX_3 和 GL_MAP1_COLOR_4，调用 glEvalCoord1()将同时产生位置和颜色。顶点求值程序每次只可以激活一个，但定义却可以同时进行。纹理求值程序也是如此。然而，求值程序还可用以产生任何顶点、法向、颜色和纹理坐标的结合数据。对于同样类型的求值程序，如果定义和激活了不止一个，则使用维数最高的。

可以利用 glEvalCoord1*()来求一个已定义和激活的一维映射的值。

void glEvalCoord1{fd}{v}(TYPE u);

此函数对已激活的一维映射求值。参量 u 是坐标范围值（在命令的向量形式中，为指向该值的指针）。

3. 定义均匀间隔的坐标值

调用 glEvalCoord1()时，u 值可以采用任何形式。最普遍的是采用均匀间隔数值，如前面例 12-1 中所示。利用 glMapGrid1*()定义一维栅格，可以得到均匀间隔数值。应用时需要用到 glEvalMesh1()。

---

void glMapGrid1{fd}(GLint n, TYPE u1, TYPE u2);

此函数定义从 u1 到 u2 分 n 步的均匀间隔栅格。

---

void glEvalMesh1(GLenum mode, GLint p1, GLint p2);

此函数将当前已定义过的栅格映射应用于所有激活的求值程序。mode 可以是 GL_POINT 或 GL_LINE，取决于需要画点还是画沿着曲线连接起来的线段。这个调用与在 p1 和 p2 之间的每一步（包括 p1，p2。0≤p1，p2≤n）发送命令 glEvalCoord1()的作用完全相同。

用程序来实现是这样的：

```
glBegin(GL_POINTS);    /* 或 glBegin(GL_LINE_STRIP); */
for (i = p1; i <= p2; i++)
   glEvalCoord1(u1 + i*(u2-u1)/n);
glEnd();
```

除了 i=0 或 i=n 之外，glEvalCoord()都直接以参数 u1 和 u2 来调用。

12.2.2 二维求值程序

二维情况下一切都与一维类似，只是所有函数必须带有两个参数 u 和 v。点、颜色、法向或纹理坐标必须应用到二维曲面上。贝塞尔曲面片的数学定义如下：

$$S(u,v)=\sum_{i=0}^{n}\sum_{j=0}^{m}B_i^n(u)B_j^m(v)P_{ij}$$

其中 P_{ij} 是一组 m×n 个控制点，并且 B_i 与一维的伯恩斯坦（Bernstein）多项式相同。如前所述，P_{ij} 可以表示顶点、法线、颜色或纹理坐标。

应用二维求值程序的过程与一维类似：

1）用 glMap2*()定义求值程序。

2）通过传递适当的数值给 glEnable()来激活求值程序。

3）在 glBegin()和 glEnd()之间通过调用 glEvalCoord2()来调用,或者通过 glMapGrid2()和

glEvalMesh2()来分别指定和应用一网格。

1. 二维求值程序的定义和求值

利用 glMap2*()和 glEvalCoord2*()可以定义并调用二维求值程序。

---

void glMap2{fd}(GLenum target, TYPEu1, TYPEu2, GLint ustride, GLint uorder, TYPEv1, TYPEv2, GLint vstride, GLint vorder, TYPE points);

此函数中，参数 target 可以是表 12-1 中任何类型的值，只是字符串 MAP1 换为 MAP2。同前面一样，这些值也可用于 glEnable()来激活相应的求值程序。u 和 v 的最小值和最大值分别表示为 u1，u2，v1，v2。参数 ustride 和 vstride 表示在这些数值的相互独立的设置之间的单精度和双精度数（视情况而定）的数目，因为这些数值允许使用者从一较大的数组之中选择一个的控制点子矩形。例如，数据若以下面形式出现：

GLfloat ctlpoints[100][100][3];

并且要用到从 ctlpoints[20][30]开始的 4×4 子集，选择 ustride 为 100×3，vstride 为 3。起始点设为&ctlpoints[20][30][0]。最后，阶数参数 uorder 和 vorder 可以不同，比如一曲面片在一个方向是三次，而在其他方向可以是二次的。

---

void glEvalCoord2{fd}{v}(TYPE u, TYPE v);

此函数对激活的二维映射进行求值。参数 u 和 v 是坐标范围值（在命令的向量形式下，为指向该值的指针）。如果顶点的任一求值程序被激活 (GL_MAP2_VERTEX_3 或 GL_MAP2_VERTEX_4)，那么将分析计算曲面的法线。通过传递 GL_AUTO_NORMAL 给 glEnable()，将激活法线的自动产生，并且将法线与顶点的产生相关联。如果自动法线产生是处在取消状态的，则采用相应的激活了的法线映射来产生法线。没有此映射，则采用当前法线。

2. 二维例子：贝塞尔曲面

如图 12-2 所示，例 12-2 利用求值程序绘制了贝塞尔线框曲面。在这个例子中，曲面由每个方向的九条曲线组成，而每条曲线由 30 段组成。为了得到整个程序，还要加上例 12-1 中的 myReshape()和 main()子程序。

例 12-2 绘制贝塞尔曲面：bezsurf.c

```
#include "glos.h"


#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
```

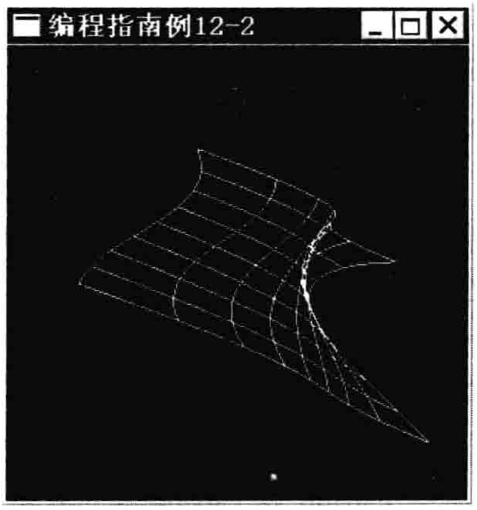


图 12-2 贝塞尔曲面

```
void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLfloat ctrlpoints[4][4][3] = {
    {{-1.5, -1.5, 4.0}, {-0.5, -1.5, 2.0},
     {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
    {{-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
     {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
    {{-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0},
     {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
    {{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
     {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};

void CALLBACK display(void)
{
    int i, j;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix ();
    glRotatef(85.0, 1.0, 1.0, 1.0);
```

```
    for (j = 0; j <= 8; j++) {
     glBegin(GL_LINE_STRIP);
        for (i = 0; i <= 30; i++)
         glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
     glEnd();
     glBegin(GL_LINE_STRIP);
        for (i = 0; i <= 30; i++)
         glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
     glEnd();
    }
    glPopMatrix ();
    glFlush();
}

void myinit(void)
{
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
        0, 1, 12, 4, &ctrlpoints[0][0][0]);
    glEnable(GL_MAP2_VERTEX_3);
    glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    if (!h) return;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
     glOrtho(-4.0, 4.0, -4.0*(GLfloat)h/(GLfloat)w,
        4.0*(GLfloat)h/(GLfloat)w, -4.0, 4.0);
    else
```

```
    glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
        4.0*(GLfloat)w/(GLfloat)h, -4.0, 4.0, -4.0, 4.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例12-2");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

3. 定义二维均匀间隔坐标值

二维情况下，glMapGrid2*()和 glEvalMesh2()命令与一维类似，只是需要同时包括 u 和 v 信息。

---

void glMapGrid2{fd}(GLint nu, TYPEu1, TYPEu2, GLint nv, TYPEv1, TYPEv2);
void glEvalMesh2(GLenum mode, GLint p1, GLint p2, GLint q2, GLint q2);

这两个命令定义二维映射的栅格，从 u1 到 u2 均匀间隔步长的 nu 步和从 v1 到 v2 的 nv 步（glMapGrid2*()），然后将这些栅格应用于所有激活的求值程序（glEvalMesh2()）。与这两个命令的一维形式相比最显著的差别是：在 glEvalMesh2()中，参数 mode 除可取 GL_POINT 或 GL_LINE 外，还可以是 GL_FILL，用四边形网格图元来产生填充多边形。准确地说，glEvalMesh2()相当于下面的三段程序代码。惟一的不同是，当 i=nu 或 j=nv 时，参数正好等于 u2 或 v2，而不是 u1+nu*(u2-u1)/nu，导致结果并不与 glEvalMesh2()完全相同。

```
glBegin(GL_POINTS);         /* mode == GL_POIN */
for (i = nu1; i <= nu2; i++)
    for (j = nv1; j <= nv2; j++)
        glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
glEnd();
```

或

```
for (i = nu1; i <= nu2; i++) {      /* mode == GL_LINE */
   glBegin(GL_LINES);
      for (j = nv1; j <= nv2; j++)
         glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
   glEnd();
  }
for (j = nv1; j <= nv2; j++)
   glBegin(GL_LINES);
   for (i = nu1; i <= nu2; i++)
      glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
   glEnd();
  }
```

或

```
for (i = nu1; i < nu2; i++) {      /* mode == GL_FILL */
   glBegin(GL_QUAD_STRIP);
   for (j = nv1; j <= nv2; j++)
      glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
      glEvalCoord2(u1 + (i+1)*(u2-u1)/nu, v1+j*(v2-v1)/nv);
   glEnd();
}
```

例 12-3　用网格绘制的、带灯光和阴影的贝塞尔曲面

```
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void initlights(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);
```

```
GLfloat ctrlpoints[4][4][3] = {
    {{-1.5, -1.5, 4.0}, {-0.5, -1.5, 2.0},
    {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
    {{-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
    {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
    {{-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0},
    {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
    {{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
    {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};

void initlights(void)
{
   GLfloat ambient[] = { 0.2, 0.2, 0.2, 1.0 };
   GLfloat position[] = { 0.0, 0.0, 2.0, 1.0 };
   GLfloat mat_diffuse[] = { 0.6, 0.6, 0.6, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[] = { 50.0 };

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);

   glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
   glLightfv(GL_LIGHT0, GL_POSITION, position);

   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
}

void CALLBACK display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
   glRotatef(85.0, 1.0, 1.0, 1.0);
```

```
    glEvalMesh2(GL_FILL, 0, 20, 0, 20);
    glPopMatrix();
    glFlush();
}

void myinit(void)
{
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glEnable (GL_DEPTH_TEST);
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
        0, 1, 12, 4, &ctrlpoints[0][0][0]);
    glEnable(GL_MAP2_VERTEX_3);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
    initlights();   /* for lighted version only */
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    if (!h) return;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho(-4.0, 4.0, -4.0*(GLfloat)h/(GLfloat)w,
        4.0*(GLfloat)h/(GLfloat)w, -4.0, 4.0);
    else
    glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
        4.0*(GLfloat)w/(GLfloat)h, -4.0, 4.0, -4.0, 4.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
```

```
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 12-3");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

例 12-3 的结果如图 12-3 所示。

图 12-3　用网格绘制带灯光和阴影的贝塞尔曲面

4. 求值程序应用于纹理

例 12-4 一次激活了两个求值程序：第一个产生与例 12-3 中一样的贝塞尔曲面上的三维点，第二个产生纹理坐标。在这种情形下，纹理坐标与曲面上的 u 和 v 坐标相同，但是必须产生一个特定贝塞尔平面片来完成这个工作。

这个平面片定义于顶点在（0，0），（0，1），（1，0）和（1，1）的正方形上；在角点（0，0）处产生（0，0），角点（0，1）处产生（0，1），如此类推。因为它是二阶的，在点（u，v）对纹理求值产生了纹理坐标（s，t）。它与顶点求值程序同时激活，所以当绘制曲面时二者都发生作用。参见彩图 19。如果希望纹理在每个方向重复三次，改变数组 texpts[ ][ ][ ]中的每一个 1.0 为 3.0。因为本例中纹理的覆盖，曲面用纹理图的九个拷贝进行渲染。

例 12-4　使用纹理的求值程序

```
#include "glos.h"

#include <GL/gl.h>
```

```
#include <GL/glu.h>
#include <GL/glaux.h>
#include <math.h>

void myinit(void);
void makeImage(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

GLfloat ctrlpoints[4][4][3] = {
    {{ -1.5, -1.5, 4.0}, { -0.5, -1.5, 2.0},
    {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
    {{ -1.5, -0.5, 1.0}, { -0.5, -0.5, 3.0},
    {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
    {{ -1.5, 0.5, 4.0}, { -0.5, 0.5, 0.0},
    {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
    {{ -1.5, 1.5, -2.0}, { -0.5, 1.5, -2.0},
    {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};

GLfloat texpts[2][2][2] = {{{0.0, 0.0}, {0.0, 1.0}},
          {{1.0, 0.0}, {1.0, 1.0}}};

void CALLBACK display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glEvalMesh2(GL_FILL, 0, 20, 0, 20);
   glFlush();
}

#define imageWidth 64
#define imageHeight 64
GLubyte image[3*imageWidth*imageHeight];
```

```
void makeImage(void)
{
    int i, j;
    float ti, tj;

    for (i = 0; i < imageWidth; i++) {
    ti = 2.0*3.14159265*i/imageWidth;
    for (j = 0; j < imageHeight; j++) {
        tj = 2.0*3.14159265*j/imageHeight;

        image[3*(imageHeight*i+j)] = (GLubyte) 127*(1.0+sin(ti));
        image[3*(imageHeight*i+j)+1] = (GLubyte) 127*(1.0+cos(2*tj));
        image[3*(imageHeight*i+j)+2] = (GLubyte) 127*(1.0+cos(ti+tj));
    }
    }
}

void myinit(void)
{
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
        0, 1, 12, 4, &ctrlpoints[0][0][0]);
    glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1, 2, 2,
        0, 1, 4, 2, &texpts[0][0][0]);
    glEnable(GL_MAP2_TEXTURE_COORD_2);
    glEnable(GL_MAP2_VERTEX_3);
    glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
    makeImage();
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, 3, imageWidth, imageHeight, 0,
        GL_RGB, GL_UNSIGNED_BYTE, image);
    glEnable(GL_TEXTURE_2D);
```

```
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_NORMALIZE);
    glShadeModel (GL_FLAT);
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    if (!h) return;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho(-4.0, 4.0, -4.0*(GLfloat)h/(GLfloat)w,
        4.0*(GLfloat)h/(GLfloat)w, -4.0, 4.0);
    else
    glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
        4.0*(GLfloat)w/(GLfloat)h, -4.0, 4.0, -4.0, 4.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glRotatef(85.0, 1.0, 1.0, 1.0);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 12-4");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

例 12-4 的运行结果如图 12-4 所示。

图 12-4　例 12-4 的运行结果

12.3　GLU 的 NURBS 接口程序

尽管求值程序是 OpenGL 中可以用来直接绘制曲线和曲面的基本工具，并在硬件中实现时显得非常有效，但它们总是需要通过应用软件的高级库进行访问。GLU 提供了建立在 OpenGL 求值程序之上的一个 NURBS 接口程序。

12.3.1　简单的 NURBS 范例

一旦理解了 NURBS，那么编写 OpenGL 代码操作 NURBS 曲线和曲面就相对容易了，包括带有灯光和纹理图。下面的这些步骤便可以绘制 NURBS 曲线或未经修剪的曲面。

如果打算使用带有光照的 NURBS 曲面，需要调用带有 GL_AUTO_NORMAL 参数的 glEnable()命令自动产生曲面的法线（也可人工计算）。

1）用 gluNewNurbsRenderer()将产生指向 NURBS 对象的指针，在创建 NURBS 曲线或曲面时作为定位基准。

2）需要时,调用 gluNurbProperty()来选择渲染的数值，比如用于渲染 NURBS 对象的多边形或线的最大尺寸。

3）如果希望有出错提示，调用 gluNurbsCallback()（错误检查将使性能略有降低）。

4）通过调用 gluBeginCurve()或 gluBeginSurface()开始曲线或曲面的绘制。

5）产生并渲染曲线或曲面。用 NURBS 对象的控制点（有理或无理的）、节点次序、基本多项式函数的阶数，调用函数 gluNurbsCurve()或 gluNurbsSurface()至少一次。指定曲面的法向或纹理坐标时可以再次调用这些函数。

6）调用 gluEndCurve()或 gluEndSurface()完成曲线或曲面。

例 12-5 采用了从-3.0 到 3.0 的控制点，以对称的丘陵状渲染了一个 NURBS 曲面。基本函数是立方 B 样条，但是节点次序是不均匀的，在每个端点处有 4 种不同，这使得基本函数在每个方向的响应类似于贝塞尔曲线。曲面被照亮，带有灰色的散射和白色的镜子一样的加亮区。图 12-5 显示了光照的线框曲面。

图 12-5　NURBS 曲面

例 12-5　绘制 NURBS 曲面

```
// 编程指南例 12-5：以对称的山丘形状绘制 NURBS 曲面

#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void init_surface(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

GLfloat ctlpoints[4][4][3];

GLUnurbsObj *theNurb;

/*
```

```
 *  将曲面的控制点初始化为一小山
 *  控制点 x， y 和 z 的变化范围为-3 到 +3
 */
void init_surface(void)
{
   int u, v;
   for (u = 0; u < 4; u++) {
   for (v = 0; v < 4; v++) {
      ctlpoints[u][v][0] = 2.0*((GLfloat)u - 1.5);
      ctlpoints[u][v][1] = 2.0*((GLfloat)v - 1.5);

      if ( (u == 1 || u == 2) && (v == 1 || v == 2))
      ctlpoints[u][v][2] = 3.0;
      else
      ctlpoints[u][v][2] = -3.0;
   }
   }
}

/*  初始化材质特性和深度缓存
 */

void myinit(void)
{
   GLfloat mat_diffuse[] = { 0.7, 0.7, 0.7, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[] = { 100.0 };

   glClearColor (0.0, 0.0, 0.0, 1.0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
```

```
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);

    init_surface();

    theNurb = gluNewNurbsRenderer();
    gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 25.0);
    gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
}

void CALLBACK display(void)
{
    GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    glRotatef(330.0, 1.,0.,0.);
    glScalef (0.5, 0.5, 0.5);

    gluBeginSurface(theNurb);
    gluNurbsSurface(theNurb,
        8, knots,
        8, knots,
        4 * 3,
        3,
        &ctlpoints[0][0][0],
        4, 4,
        GL_MAP2_VERTEX_3);
    gluEndSurface(theNurb);

    glPopMatrix();
    glFlush();
```

```
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    if (!h) return;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLdouble)w/(GLdouble)h, 3.0, 8.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (0.0, 0.0, -5.0);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 12-5");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

如例 12-5 所示，gluNewNurbsRenderer()返回一个新的 NURBS 对象，其类型为指向 GLUnurbsObj 结构的指针。gluBeginSurface()和 gluEndSurface()这一对函数将渲染子函数括在一起，保存和恢复求值程序状态。这三个函数在附录 3 中进行了总结。两个稍复杂的函数 gluNurbsProperty()和 gluNurbsSruface()将在本节讨论。

---

void gluNurbsProperty(GLUnurbsObj *nobj, GLenum property, GLfloat value);

此函数控制 NURBS 对象的属性 nobj。参数 property 指定了性质，可以是 GLU_SAMPLING_TOLERANCE，GLU_DISPLAY_MODE，GLU_CULLING 或 GLU_AUTO_LOAD_MATRIX。参数 value 指出性质是什么。NURBS 对象作为图元进行了渲染，对其参

数（u 和 v）的不同数值进行了采样，并且分成小的线段或多边形以供渲染。GLU_SAMPLING_TOLERANCE 控制了这个 NURBS 对象的采样周期。缺省值 50.0 意味着最大的采样线段或采样多边形的边为 50.0 个像素。

GLU_DISPLAY_MODE 的缺省值是 GLU_FILL，将曲面作为多边形进行渲染。如果以 GLU_OUTLINE_POLYGON 作为显示模式属性，则渲染出多边形轮廓。最后，GLU_OUTLINE_PATCH 渲染曲面片的轮廓并修剪曲线（参见下一节）。

如果 NURBS 对象完全落在视图体以外，GLU_CULLING 将不执行网格化，从而提高了性能；设置此属性为 GL_TRUE（缺省为 GL_FALSE），激活消隐（culling）。属性 GLU_AUTO_LOAD_MATRIX 决定是否将投影矩阵，模型视图矩阵和视区从 OpenGL 服务器下载（缺省为 GL_TRUE），或者应用程序是否必须与 gluLoadSamplingMatrices()（GL_FALSE）一起提供这些矩阵。

---

void gluNurbsSurface (GLUnurbsObj *nobj, GLint uknot_count, GLfloat *uknot, GLint vknot_count, GLfloat *vknot, GLint u_stride, GLint v_stride, GLfloat *ctlarray, GLint uorder, GLint vorder, GLenum type);

此函数描述 NURBS 曲面（或曲面法向或纹理坐标）的顶点 nobj。u 和 v 的参量方向必须指定几个值，如节点次序（uknot 和 vknot），节点数目（ukont_count 和 vknot_count），NURBS 曲面的多项式阶数（uorder 和 vorder）。注意在这里并未指定控制点的数目，而是在节点数目减去阶数时，通过决定沿着每个参数的控制点的数量得到。然后，曲面控制点的数量等于在每个参数方向上控制点数目的乘积。参数 ctlarray 指向控制点数组。

最后一个参数 type 指定二维求值程序的类型。一般情况下，可以分别用 GL_MAP2_VERTEX_3 表示无理的、用 GL_MAP2_VERTEX_4 表示有理的控制点。也可以用其他类型，如 GL_MAP2_TEXTURE_COORD_* 或 GL_MAP2_NORMAL 来计算纹理坐标及曲面法线并赋值。

参数 u_stride 和 v_stride 代表沿每个参数方向的控制点间的浮点数的数量。求值程序的类型同它的阶数一样，影响 u_stride 和 v_stride 的值。在例 12-5 中，u_stride 为 12（4×3）——每个顶点有三个坐标（由 GL_MAP2_VERTEX_3 设置），沿参数 v 方向有四个控制点；v_stride 为 3，因为每个矩阵有 3 个坐标，并且 v 个控制点之间很接近。

绘制 NURBS 曲线与绘制曲面类似，只是所有的计算只用到一个参数 u。同样，其他函数据在 gluBeginCurve()h 和 gluEndCurve()命令对之间。

---

void gluNurbsCurve (GLUnurbsObj *nobj, GLint uknot_count, GLfloat * uknot, GLint u_stride, GLfloat *ctlarray, GLint uorder, GLenum type);

此函数为对象 nobj 定义了一个 NURBS 曲线。其中的参数与 gluNurbsSurface()的参数具有相同的含义。注意这个程序仅需要一个节点次序和一个 NURBS 对象阶数的声明。如果曲线在 gluBeginCurve()/gluEndCurve()之间定义，那么 type 可以是一维求值程序的任何有效类型（如 GL_MAP1_VERTEX_3 或 GL_MAP1_VERTEX_4）。

12.3.2 修剪

为了用 OpenGL 创建一个修剪过的 NURBS 曲面，先创建一个未经修剪的曲面。在调用了 gluBeginSurface()和 gluNurbsSurface()之后，未调用 gluEndSurface()之前，调用 gluBeginTrim()开始修剪。可以创建的修剪曲线有两种：用 gluPwlCurve()产生分段的线性曲线，或用 gluNurbsCurve()产生 NURBS 曲线。分段线性曲线实际上是一系列直线。待修剪的 NURBS 曲线必须位于参数的（u，v）空间中的单位正方形内。NURBS 修剪曲线的类型通常是 GLU_MAP1_TRIM2。有时也用到类型 GLU_MAP1_TRIM3，这里曲线在二维齐次空间 (u', v', w')内通过(u, v) = (u'/w', v'/w')来描述。

---

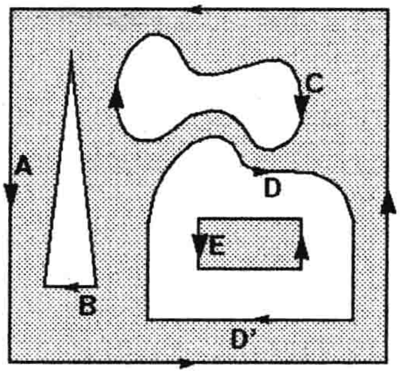
void gluPwlCurve (GLUnurbsObj *nobj, GLint count, GLfloat *array, GLint stride, GLenum type);

此函数为 NURBS 对象 nobj 描述了一条分段线性的修剪曲线。曲线上有 count 个点，以数组形式给出。type 可以是 GLU_MAP1_TRIM_2（最为普遍）或 GLU_MAP1_TRIM_3（(*u, v, w*) 齐次参数空间）。type 决定下一个顶点的浮点数的数量 stride 是 2 还是 3。

进行修剪操作时，程序员需要考虑修剪曲线的方向是逆时针还是顺时针，从而确保留下了需要的部分曲面。假想沿着曲线行走，左侧包含于曲面内，右侧将修剪掉。例如，修剪一个逆时针组成的闭合回路，回路内的所有物体都被包含进来。如果修剪两个不相交的逆时针回路组成的内部区域，两个回路之间的所有物体被包含进来。如果一个逆时针回路的内部有两个逆时针回路，那么修剪后的区域有两个洞。最外层的修剪曲线必须是逆时针的。一般地，可以取绕整个单位正方形的修剪曲线，这样就包含了其内部的所有物体，这也是不指定修剪曲线时的缺省情况。

修剪曲线必须是封闭的、不相交的。只要曲线的端点可以连接成闭合曲线，便可以将几条修剪曲线结合起来。还可以嵌入曲线，创建漂浮在空间中的“岛屿”。曲线的方向需要注意。例如，如果指定两个逆时针曲线形成修剪区域，其中一个在另一个内部，将得到错误的结果。两条曲线之间的区域在一条曲线的左侧，在另一条曲线的右侧，从而无法判定究竟是否包含在内。图 12-6 举例说明了一些正确的情况。

图 12-7 例示了与图 12-5 相同的小山丘，这次带有由分段线性的曲线和 NURBS 曲线联合组成的修剪曲线。产生这个图的程序类似于例 12-5；其中的差异在例 12-6 中加以说明。




```
gluBeginSurface();
   gluNurbsSurface(...);
   gluBeginTrim();
      gluPwlCurve(...); /* A */
   gluEndTrim();
   gluBeginTrim();
      gluPwlCurve(...); /* B */
   gluEndTrim();
   gluBeginTrim();
      gluNurbsCurve(...);/* C */
   gluEndTrim();
   gluBeginTrim();
      gluNurbsCurve(...);/* D */
      gluPwlCurve(...); /* D' */
   gluEndTrim();
   gluBeginTrim();
      gluPwlCurve(...); /* E */
   gluEndTrim();
gluEndSurface();
```

图 12-6　参数修剪曲线

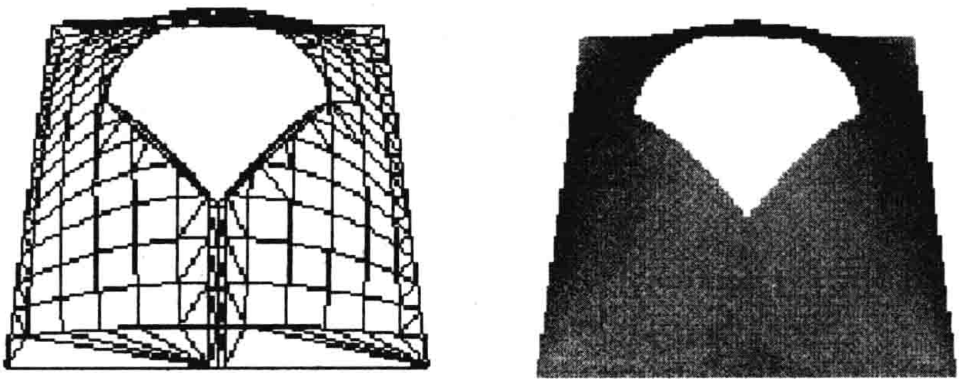


图 12-7　修剪过的 NURBS 曲面

例 12-6　修剪 NURBS 曲面

```
void myinit(void)
{
GLfloat mat_diffuse[] = { 0.6, 0.6, 0.6, 1.0 };
GLfloat mat_specular[] = { 0.9, 0.9, 0.9, 1.0 };
GLfloat mat_shininess[] = { 128.0 };
glClearColor (0.0, 0.0, 0.0, 1.0);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

```
glDepthFunc(GL_LEQUAL);
glEnable(GL_DEPTH_TEST);
glEnable(GL_AUTO_NORMAL);
glEnable(GL_NORMALIZE);
init_surface();
theNurb = gluNewNurbsRenderer();
gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 50.0);
gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
}

void display(void)
{
GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
GLfloat edgePt[5][2] = /* 逆时针 */
{{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0}, {0.0, 0.0}};
GLfloat curvePt[4][2] = /* 顺时针 */
{{0.25, 0.5}, {0.25, 0.75}, {0.75, 0.75}, {0.75, 0.5}};
GLfloat curveKnots[8] =         {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
GLfloat pwlPt[4][2] = /* clockwise */
{{0.75, 0.5}, {0.5, 0.25}, {0.25, 0.5}};
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glPushMatrix();
glRotatef(330.0, 1.,0.,0.);
glScalef (0.5, 0.5, 0.5);
gluBeginSurface(theNurb);
gluNurbsSurface(theNurb, 8, knots, 8, knots,
             4 * 3, 3, &ctlpoints[0][0][0],
             4, 4,  GL_MAP2_VERTEX_3);
gluBeginTrim (theNurb);
gluPwlCurve(theNurb, 5, &edgePt[0][0], 2, GLU_MAP1_TRIM_2);
gluEndTrim (theNurb);
gluBeginTrim (theNurb);
gluNurbsCurve(theNurb, 8, curveKnots, 2, &curvePt[0][0], 4, GLU_MAP1_TRIM_2);
gluPwlCurve (theNurb, 3, &pwlPt[0][0], 2, GLU_MAP1_TRIM_2);
gluEndTrim (theNurb);
```

```
gluEndSurface(theNurb);
glPopMatrix();
glFlush();
}
```

在例 12-6 中，gluBeginTrim()和 gluEndTrim()将每条修剪曲线括在其中。第一次修剪，在由数组 edgePt[][]定义的顶点上进行，绕整个参数空间的单位正方形逆时针旋转。如果物体不会被内部的顺时针修剪曲线切去，便可以保证所有的物体都能画出。第二次修剪是 NURBS 修剪曲线和分段线性修剪曲线的综合作用。NURBS 曲线在端点（0.9,0.5）和（0.1,0.5）处与分段线性曲线相交，形成一条封闭的顺时针曲线。

第 13 章　选择机制

本章主题

在实际应用中，仅有部分 OpenGL 图形程序只需要简单画出二维或三维物体的静态图像，而其他一些应用程序则要求用户识别屏幕上的物体，并对其进行移动、修改、删除或其他操作。实际上 OpenGL 就是为支持这种交互式应用程序而设计的。由于画在屏幕上的物体通常都经历过多次旋转、平移和投影变换，在三维场景中要确定用户所选择的究竟是哪一个物体就显得十分困难。

令人高兴的是，OpenGL 提供了一种选择机制，自动地通知用户在窗口的某个特定区域里画出的是哪一个物体。用户可以结合使用选择机制和特殊的实用程序来确定用户鼠标指定或拾取的是哪一个物体。

选择实际上是 OpenGL 的一种操作模式；反馈是另一种模式。在反馈模式中，程序员可以利用图形硬件和 OpenGL 来完成通常的渲染计算。然而 OpenGL 并不是利用计算结果将图像绘制到屏幕上，而是将绘图信息返回（或称反馈）给用户。如果用户要将三维物体绘制在绘图仪而不是屏幕上，则需要将各项物体以反馈模式绘出，搜集绘图指令，然后将它们转换为绘图仪可以接受的命令。

在选择和反馈模式中，绘图信息将返回给应用程序，而不像在渲染模式中那样发送到帧缓存中。因此，当 OpenGL 处在选择或反馈模式下时，不会发生屏幕绘图。本章分别解释了这两种模式。

- “选择”，讨论了如何使用选择模式及相关函数来允许应用程序的用户拾取屏幕物体。
- “反馈”，描述了如何获取绘图信息及信息格式。

13.1　选择

通常情况下，若使用 OpenGL 的选择机制，首先要将场景绘制到帧缓存中，然后进入选择模式重画场景。然而一旦处在了选择模式下，在用户退出该模式之前，帧缓存的内容将不发生变化。在退出选择模式时，OpenGL 返回一列与视图体相交的图元。与视图体相交的每个图元都会产生一个选中信息。图元列实际上是以整型值名称及相关数据（选中记录）的数组形式返回的。在选择模式下发送图元绘图命令时，程序员可以通过加载名称的方式来创建命名堆栈。因此，在返回名称列表时，可以用它来确定用户选中了哪一个物体。

除了选择机制，OpenGL 还提供了一个实用程序来简化某些场合下的选择。一般来说，程序员可以利用这个程序来确定哪个物体画在光标附近，这样便于用户识别拾取的究竟是哪一个物体。还可以通过定义附加裁剪平面来限制选择区域。由于拾取是选择机制的一种特殊情况，因此我们先讨论选择机制，然后再阐述拾取。

13.1.1　基本步骤

使用选择机制的基本步骤如下：

1）用函数 glSelectBuffer()定义用于记录返回的选中记录的数组。

2）将函数 glRenderMode()的参数设置为 GL_SELECT，进入选择模式。

3）用函数 glInitNames()和 glPushName()初始化命名堆栈。

4）定义用于选择的视图体。通常此视图体与用于绘制场景的视图体不同，所以用函数 glPushMatrix()和 glPopMatrix()保存和恢复当前变换状态。

5）交替发送图元绘制命令以及命名堆栈操作命令，为相交的图元分配合适的名称。

6）退出选择模式，处理返回的选择数据（选中记录）。

下面对函数 glSelectBuffer()和 glRenderMode()进行描述。

---

void glSelectBuffer(GLsizei size, GLuint *buffer);

指定用于返回的选中数据的数组。参数 buffer 是一指针，指向无符号整型数组，返回数据就存放在该数组里；参数 size 表示数组中可以存储数据的最大数量。在进入选择模式之前还需要调用 glSelectBuffer()命令。

GLint glRenderMode(GLenum mode);

控制应用程序的渲染模式：渲染、选择或反馈模式中的一种。参数 mode 可以为 GL_RENDER（缺省）、GL_SELECT 或 GL_FEEDBACK。如果不以其他模式调用函数 RenderMode()，那么应用程序将保持给定的模式。在进入选择模式之前，必须调用函数 glSelectBuffer()以定义选择数组。类似地，在进入反馈模式之前，要调用函数 glFeedbackBuffer()来定义反馈数组。只有当前的渲染模式是（注意不是 mode 参数）GL_SELECT 或 GL_FEEDBACK 时，glRenderMode()的返回值才有意义：返回值是选中的数量，在退出相应的模式时，返回值为反馈数组中的数值；如果值为负，则意味着选择数组或是反馈数组溢出。用参数 GL_RENDER_MODE 调用函数 glGetIntegerv()可以获得当前的模式。

13.1.2　创建命名堆栈

如前所述，命名堆栈构成了选择信息的基础。为了创建命名堆栈，首先需要用 glInitNames()

命令对其进行初始化，简单地将堆栈清空，然后在发出相应的绘图命令时加上整型的命名。堆栈操作命令可以将一个命名压入堆栈（使用 glPushName()命令），也可将一个命名弹出（使用 glPopName()命令），或将栈顶命名替换为别的名称（使用 glLoadName()命令）。例 13-1 显示了使用这些命令时命名堆栈的操作代码。

例 13-1 创建命名堆栈

```
glInitNames();
glPushName(-1);
glPushMatrix();   /* 保存当前变换状态*/
glLoadName(1);
drawSomeObject();
glLoadName(2);
drawAnotherObject();
glLoadName(3);
    drawYetAnotherObject();
    drawJustOneMoreObject();
glPopMatrix ();   /*恢复先前的变换状态*/
```

在这个例子中，要画出的前两个物体有其自己的名称，而第三个和第四个物体共用一个名称。在这种设置下，如果在第三个和/或第四个物体上发生了选择事件，都只返回一个选中记录。

---

void glInitNames(void);

清空命名堆栈。

---

void glPushName(GLuint name);

将 name 压入命名堆栈。如果压入的命名已经超出了堆栈容量，则产生错误 GL_STACK_OVERFLOW。在不同的 OpenGL 实现中，命名堆栈的深度可以是不同的，但至少为 64。参数为 GL_NAME_STACK_DEPTH 的 glGetIntegerv()命令可以用来获取命名堆栈的深度。

---

void glPopName(void);

弹出栈顶命名。对一空堆栈进行弹出操作将产生 GL_STACK_UNDERFLOW 错误。

void glLoadName(GLuint name);

以 name 替换命名堆栈的栈顶值。若在命令 glInitNames()之后调用该命令，因为堆栈为空，将产生 GL_INVALID_OPERATION 错误。要避免这种错误，如果堆栈初始为空，那么在调用 glLoadName()之前至少应调用一次 glPushName()，将名称压入命名堆栈。

如果没有进入选择模式，则 glPushName()，glPopName()和 glLoadName()调用将被忽略。在绘图代码中贯穿使用这些函数，会使代码变得简单。

13.1.3　选中记录

在选择模式下，与视图体相交的图元会产生一个选中信息。无论何时执行命名堆栈操作命令，或调用 glRenderMode()函数，如果对堆栈进行了操作或调用了 glRenderMode()函数，并确实有选中事件，OpenGL 都将选中记录写入选择数组。通过这一处理，享有同一名称的物体——例如，由多于一个图元组成的物体——就不会产生多个选中记录。另外，如果未调用 glRenderMode()，则无法保证选中记录被写入数组。

注意：在图元之外，由 glRasterPos()产生的有效坐标也会产生一个选中事件。

每一个选中记录包括四个部分，依次为：

- 选中时命名堆栈中名称的数量。
- 所有与视图体相交的图元的所有顶点中，最大和最小的窗口 Z 坐标值。这两个值都在[0，1]范围内，都要乘以 2^{32}-1，并舍入为最接近的整数值。
- 选中时命名堆栈中的内容，最底部的元素在最前面；
- 在进入选择模式时，OpenGL 要初始化指向选择数组开始位置的指针。在选中记录被写入数组时，该指针都相应地进行更新。如果写入的选中记录导致了数组元素超界（超出了函数 glSelectBuffer()的参数 size 定义的个数），OpenGL 将设置一个溢出标志。

13.1.4　一个选择的实例

在例 13-2 中，要在选择模式下绘制四个三角形（一个绿色、一个红色和两个黄色），然后处理相应的选中记录。第一个三角形产生一次选中事件，第二个则不产生，第三和第四个只产生一个选中事件。在本例中，分别定义了绘制三角形的函数（drawTriangle()）及绘制表示视图体的方盒线框图的函数（drawViewVolume()）。函数 processHits()用于打印选择数组。最后，selectObjects()在选择模式下绘制三角形，并产生选中记录。

例 13-2　一个选择的实例

```
#include "glos.h"
```

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <stdio.h>

#include <math.h>

void myinit(void);
void drawTriangle (GLfloat x1, GLfloat y1, GLfloat x2,
    GLfloat y2, GLfloat x3, GLfloat y3, GLfloat z);
void drawViewVolume (GLfloat x1, GLfloat x2, GLfloat y1,
    GLfloat y2, GLfloat z1, GLfloat z2);
void drawScene (void);
void processHits (GLint hits, GLuint buffer[]);
void selectObjects(void);
void CALLBACK display(void);

/ *
 * 在距离原点 z 单位处绘制一个顶点为(x1, y1), (x2, y2), (x3, y3)的三角形
 * /
void drawTriangle (GLfloat x1, GLfloat y1, GLfloat x2,
    GLfloat y2, GLfloat x3, GLfloat y3, GLfloat z)
{
    glBegin (GL_TRIANGLES);
    glVertex3f (x1, y1, z);
    glVertex3f (x2, y2, z);
    glVertex3f (x3, y3, z);
    glEnd ();
}

/ *
 * 以这些 x, y, z 值绘制一个矩形盒子
 * /

void drawViewVolume (GLfloat x1, GLfloat x2, GLfloat y1,
```

```
    GLfloat y2, GLfloat z1, GLfloat z2)
{
    glColor3f (1.0, 1.0, 1.0);
    glBegin (GL_LINE_LOOP);
    glVertex3f (x1, y1, -z1);
    glVertex3f (x2, y1, -z1);
    glVertex3f (x2, y2, -z1);
    glVertex3f (x1, y2, -z1);
    glEnd ();

    glBegin (GL_LINE_LOOP);
    glVertex3f (x1, y1, -z2);
    glVertex3f (x2, y1, -z2);
    glVertex3f (x2, y2, -z2);
    glVertex3f (x1, y2, -z2);
    glEnd ();

    glBegin (GL_LINES);
    glVertex3f (x1, y1, -z1);
    glVertex3f (x1, y1, -z2);
    glVertex3f (x1, y2, -z1);
    glVertex3f (x1, y2, -z2);
    glVertex3f (x2, y1, -z1);
    glVertex3f (x2, y1, -z2);
    glVertex3f (x2, y2, -z1);
    glVertex3f (x2, y2, -z2);
    glEnd ();
}

/ *
 * drawScene() 绘制 4 个三角形和一个线框代表视图体.
 * /
void drawScene (void)
{
    glMatrixMode (GL_PROJECTION);
```

```
    glLoadIdentity ();
    gluPerspective (40.0, 4.0/3.0, 0.01, 100.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
    gluLookAt (7.5, 7.5, 12.5, 2.5, 2.5, -5.0, 0.0, 1.0, 0.0);
    glColor3f (0.0, 1.0, 0.0);  /*  绿色三角形  */
    drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -2.0);
    glColor3f (1.0, 0.0, 0.0);  /*  红色三角形  */
    drawTriangle (2.0, 7.0, 3.0, 7.0, 2.5, 8.0, -5.0);
    glColor3f (1.0, 1.0, 0.0);  /*  两个黄色三角形  */
    drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, 0.0);
    drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -10.0);
    drawViewVolume (2.4, 2.5, 2.4, 2.5, 0.0, 10.0);
}

/ *
 * processHits() 打印输出选择数组的内容。
 * /
void processHits (GLint hits, GLuint buffer[])
{
    int i, j;
    GLint names, *ptr;

    printf ("hits = %d\n", hits);
    ptr = (GLuint *) buffer;
    for (i = 0; i < hits; i++)
        names = *ptr;
        printf (" number of names for hit = %d\n", names); ptr++;
        printf ("  z1 is %f;", (*ptr)/(pow(2.0,32.0)-1 )); ptr++;
        printf (" z2 is %f\n", (*ptr)/(pow(2.0,32.0)-1 )); ptr++;
        printf ("   the name is ");

        for (j = 0; j < names; j++) {
            printf ("%d ", *ptr); ptr++;
```

```
        }
        printf ("\n");
    }
}

#define BUFSIZE 512

void selectObjects(void)
{
    GLuint selectBuf[BUFSIZE];
    GLint hits;

    glSelectBuffer (BUFSIZE, selectBuf);
    (void) glRenderMode (GL_SELECT);

    glInitNames();
    glPushName((unsigned)-1);

    glPushMatrix ();
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho (2.4, 2.5, 2.4, 2.5, 0.0, 10.0);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
    glLoadName(1);
    drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -2.0);
    glLoadName(2);
    drawTriangle (2.0, 7.0, 3.0, 7.0, 2.5, 8.0, -5.0);
    glLoadName(3);
    drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, 0.0);
    drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -10.0);
    glPopMatrix ();
    glFlush ();

    hits = glRenderMode (GL_RENDER);
```

```
    processHits (hits, selectBuf);
}

void myinit (void)
{
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}

void CALLBACK display(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    drawScene ();
    selectObjects ();
    glFlush();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 13-2");
    myinit ();
    auxMainLoop(display);
    return(0);
}
```

程序运行结果如图 13-1 所示。

13.1.5 拾取

作为选择过程的扩充，我们还可以利用选择模式来确定物体是否被拾取。要实现拾取操作，可以用一个特殊的拾取矩阵与投影矩阵相结合，将绘画限制在视区中一个小区域内，并且使该区域靠近光标。然后允许某种形式的输入，如按下鼠标键等来初始化选择模式。在建立选择模式并使用了特殊的拾取矩阵后，在光标附近画出的物体就会产生选中记录。因此，

在拾取过程中经常需要确定哪个物体是画在光标附近的。

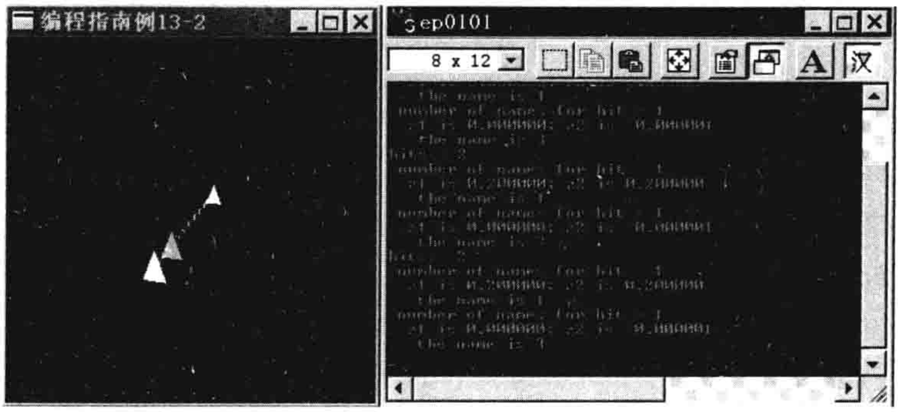


图 13-1　选择模式实例运行结果

拾取的设置与常规的选择模式差不多，主要的区别如下：

- 拾取通常是由输入设备激发的。在下例的代码中，单击鼠标左键将激发拾取函数。
- 在拾取中，需要利用实用程序 gluPickMatrix()使一特殊的投影矩阵与当前矩阵相乘。该程序应在投影矩阵与堆栈相乘之前调用。

另外一种完全不同的拾取方法是利用颜色值来辨别物体的不同组成部分。

---

void gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble height, GLint viewport[4]);

创建投影矩阵，将绘画限制在视区的一个小区域内，并用该矩阵与当前矩阵堆栈相乘。拾取区域的中心为窗口坐标下的(x, y)，通常是光标位置。width 和 height 共同定义了屏幕坐标下拾取区域的大小。(可以认为这个宽度和高度是拾取设备的灵敏度)。viewport[ ]表示当前的视区边界，该数组值可以通过调用 glGetIntegerv(GL_VIEWPORT, GLint *viewport)来得到。

由 gluPickMatrix()所创建的矩阵的最终结果是将裁剪区域转换为单位立方体-1×(x, y, z)×1（或-w×(wx, wy, wz)×w）。拾取矩阵有效地执行一次正交变换，将单位立方体的子区域映射为单位立方体。由于变换是任意的，因此可以对不同类型的区域进行拾取操作——例如，对窗口的经过旋转的矩形部分。

在某些情况下，利用附加裁剪平面来定义拾取区域会更加简单。

例 13-3 说明了一个简单的拾取操作。同时该例也演示了如何使用多个名称来识别一个图元中的不同部分。程序绘制了一个 3×3 的方格，每个方格有不同的颜色。数组 board[3][3]保存了每个格子的当前的蓝色量。按下鼠标左键，调用 pickSquares()函数来识别所拾取的是哪一个方格。每个方块由两个名称来识别——一个识别行，另一个识别列。鼠标左键按下时，选中方格的颜色会发生变化。

例 13-3 一个拾取的例子

```
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <stdio.h>

void myinit(void);
void processHits (GLint hits, GLuint buffer[]);
void CALLBACK pickSquares(AUX_EVENTREC *event);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

int board[3][3];    /* 每一个方框所能变化的颜色数量 */

void myinit(void)
{
   int i, j;
   for (i = 0; i < 3; i++)
   for (j = 0; j < 3; j ++)
      board[i][j] = 0;
   glClearColor (0.0, 0.0, 0.0, 0.0);
}

void drawSquares(GLenum mode)
{
   GLuint i, j;
   for (i = 0; i < 3; i++) {
   if (mode == GL_SELECT)
      glLoadName (i);
   for (j = 0; j < 3; j ++) {
      if (mode == GL_SELECT)
      glPushName (j);
      glColor3f ((GLfloat) i/3.0, (GLfloat) j/3.0,
```

```
            (GLfloat) board[i][j]/3.0);
        glRecti (i, j, i+1, j+1);
        if (mode == GL_SELECT)
        glPopName ();
    }
    }
}

void processHits (GLint hits, GLuint buffer[])
{
    int i, j;
    GLint ii, jj, names, *ptr;

    ptr = (GLuint *) buffer;
    for (i = 0; i < hits; i++)
    names = *ptr;
    ptr++;
    ptr++;
    ptr++;
    for (j = 0; j < names; j++)
        if (j == 0)
        ii = *ptr;
        else if (j == 1)
        jj = *ptr;
        ptr++;
    }
    board[ii][jj] = (board[ii][jj] + 1) % 3;
    }
}

#define BUFSIZE 512

void CALLBACK pickSquares(AUX_EVENTREC *event)
{
    GLuint selectBuf[BUFSIZE];
```

```
    GLint hits;
    GLint viewport[4];
    int x, y;

    x = event->data[AUX_MOUSEX];
    y = event->data[AUX_MOUSEY];
    glGetIntegerv (GL_VIEWPORT, viewport);

    glSelectBuffer (BUFSIZE, selectBuf);
    (void) glRenderMode (GL_SELECT);

    glInitNames();
    glPushName((unsigned)-1);

    glMatrixMode (GL_PROJECTION);
    glPushMatrix ();
    glLoadIdentity ();
/ *
 * 在光标附近创建一 5x5 的像素拾取区域
 * /
    gluPickMatrix ((GLdouble) x, (GLdouble) (viewport[3] - y),
    5.0, 5.0, viewport);
    gluOrtho2D (0.0, 3.0, 0.0, 3.0);
    drawSquares (GL_SELECT);

    glMatrixMode (GL_PROJECTION);
    glPopMatrix ();
    glFlush ();

    hits = glRenderMode (GL_RENDER);
    processHits (hits, selectBuf);
}

void CALLBACK display(void)
{
```

```
    glClear(GL_COLOR_BUFFER_BIT);
    drawSquares (GL_RENDER);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D (0.0, 3.0, 0.0, 3.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 13-3");
    myinit ();
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, pickSquares);
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

程序运行结果如图 13-2 所示。

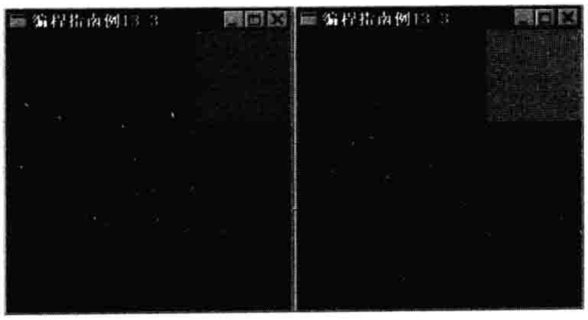


图 13-2　拾取实例运行结果

13.1.6 用多个命名和分层模型进行拾取

多命名还可用于选择场景中部分具有层次的物体。例如，如果要渲染一条汽车的装配线，我们可以这样假定，用户要选择装配线上第三辆汽车的左前方轮胎上的第三个螺钉。可以使用不同的命名来识别分层结构中的每一个层次：哪一辆汽车，哪一个轮胎，然后是哪一个螺钉。这样的例子还有很多，例如可以用一个命名来描述一个分子，而用另一个命名来区别分子中各个不同的原子。

例 13-4 在例 4-4 的基础上作了修改。该例绘制了一辆有四个轮胎的汽车，每个轮胎上有五个相同的螺钉。在例子加入了利用物体分层结构对命名堆栈进行操作的代码。

例 13-4 创建多个命名

```
draw_wheel_and_bolts()
{
long i;
draw_wheel_body();
for (i = 0; i < 5; i++)
   glPushMatrix();
      glRotate(72.0*i, 0.0, 0.0, 1.0);
      glTranslatef(3.0, 0.0, 0.0);
      glPushName(i);
         draw_bolt_body();
      glPopName();
   glPopMatrix();
  }
}

draw_body_and_wheel_and_bolts()
{
draw_car_body();
glPushMatrix();
   glTranslate(40, 0, 20);  /* 车轮 1 的位置*/
   glPushName(1);          /* 对车轮 1 命名*/
       draw_wheel_and_bolts();
   glPopName();
glPopMatrix();
```

```
glPushMatrix();
   glTranslate(40, 0, -20); /* 车轮 2 的位置 */
   glPushName(2);          /* 对车轮 2 命名*/
       draw_wheel_and_bolts();
   glPopName();
glPopMatrix();      /* 类似地，绘制最后两个车轮*/
}
```

例 13-5 用例 13-4 中的子程序来绘制三辆不同的汽车。

例 13-5 使用多个命名

```
draw_three_cars()
{
glInitNames();
glPushMatrix();
   translate_to_first_car_position();
   glPushName(1);
       draw_body_and_wheel_and_bolts();
   glPopName();
glPopMatrix();

glPushMatrix();
   translate_to_second_car_position();
   glPushName(2);
       draw_body_and_wheel_and_bolts();
   glPopName();
glPopMatrix();

glPushMatrix();
   translate_to_third_car_position();
   glPushName(3);
       draw_body_and_wheel_and_bolts();
   glPopName();
glPopMatrix();
}
```

假设还进行了拾取操作，下面是一些可能的命名堆栈的返回值及其解释。在这几个例子中，最多只返回一个选中记录：其中 d1 和 d2 是深度值。

empty	在所有的车之外进行拾取操作
2 d1 d2 2 1	第二辆车，第一个轮胎
1 d1 d2 3	第三辆车的车体
3 d1 d2 1 1 0	第一辆车的第一个轮胎上的第 0 个螺钉

上面最后一行解释假定螺钉和轮胎并不在同一个拾取区域。用户可以同时拾取车轮和螺钉。如果接收到多个选中事件，就需要决定应该处理哪一次选中事件，也许是通过深度值来决定所拾取的物体中哪一个最靠近视点。

13.1.7 拾取和深度值

例 13-6 演示了在拾取时如何使用深度值来决定拾取的是哪一个物体。该程序在正常的渲染模式下绘制了三个重叠的矩形。按下鼠标左键，则调用 pickRects()子程序。这个子程序返回光标位置，进入选定模式，初始化命名堆栈，并将拾取矩阵与堆栈相乘，然后再乘以正交投影矩阵。按下鼠标左键时，光标所在的每个矩形都发生一次选中事件。最后，要检查选择缓存的内容，来识别哪一个命名物体在光标附近的拾取区域中。

程序中的矩形是以不同的深度（或 z）值画出的。由于只用了一个命名来识别所有的三个矩形，因此只能记录下一次选中记录。当然，如果有多个矩形被拾取，那么选中时间就存在不同的最大和最小深度值。

例 13-6 用深度值进行拾取

```
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <stdio.h>

void myinit(void);
void drawRects(GLenum mode);
void processHits (GLint hits, GLuint buffer[]);
void CALLBACK pickRects(AUX_EVENTREC *event);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

void myinit(void)
```

```
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glDepthFunc(GL_LESS);
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_FLAT);
   glDepthRange (0.0, 1.0);
}

void drawRects(GLenum mode)
{
   if (mode == GL_SELECT)
   glLoadName (1);
   glBegin (GL_QUADS);
   glColor3f (1.0, 1.0, 0.0);
   glVertex3i (2, 0, 0);
   glVertex3i (2, 6, 0);
   glVertex3i (6, 6, 0);
   glVertex3i (6, 0, 0);
   glColor3f (0.0, 1.0, 1.0);
   glVertex3i (3, 2, -1);
   glVertex3i (3, 8, -1);
   glVertex3i (8, 8, -1);
   glVertex3i (8, 2, -1);
   glColor3f (1.0, 0.0, 1.0);
   glVertex3i (0, 2, -2);
   glVertex3i (0, 7, -2);
   glVertex3i (5, 7, -2);
   glVertex3i (5, 2, -2);
   glEnd ();
}

void processHits (GLint hits, GLuint buffer[])
{
   int i;
   GLint names, *ptr;
```

```
    ptr = (GLuint *) buffer;
    for (i = 0; i < hits; i++)
        names = *ptr;
    }
}

#define BUFSIZE 512

void CALLBACK pickRects(AUX_EVENTREC *event)
{
    GLuint selectBuf[BUFSIZE];
    GLint hits;
    GLint viewport[4];
    int x, y;

    x = event->data[AUX_MOUSEX];
    y = event->data[AUX_MOUSEY];
    glGetIntegerv (GL_VIEWPORT, viewport);

    glSelectBuffer (BUFSIZE, selectBuf);
    (void) glRenderMode (GL_SELECT);

    glInitNames();
    glPushName((unsigned)-1);

    glMatrixMode (GL_PROJECTION);
    glPushMatrix ();
    glLoadIdentity ();
    gluPickMatrix ((GLdouble) x, (GLdouble) (viewport[3] - y),
    5.0, 5.0, viewport);
    glOrtho (0.0, 8.0, 0.0, 8.0, -0.5, 2.5);
    drawRects (GL_SELECT);
    glPopMatrix ();
    glFlush ();
```

```
    hits = glRenderMode (GL_RENDER);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    drawRects (GL_RENDER);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho (0.0, 8.0, 0.0, 8.0, 0.0, 2.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 13-6");
    myinit ();
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, pickRects);
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

程序运行结果如图 13-3 所示。

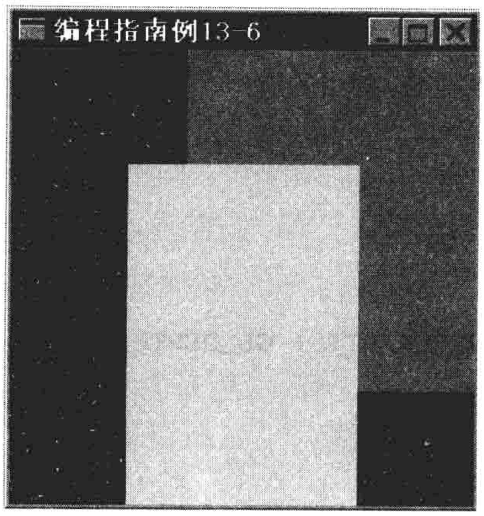


图 13-3 用深度值进行拾取实例运行结果

试一试

- 对例 13-6 进行修改，加入 glPushName()调用，在进行选择时将多个命名压入堆栈中。考虑一下此时选择缓存中的内容。
- 缺省情况下，glDepthRange()函数将 z 值映射到区间[0.0，1.0]中。修改 glDepthRange()的值，看看它如何影响在选择数组中返回的深度值。

13.1.8 在编程中使用选择机制的一些提示

多数程序允许用户交互地编辑几何体。对二维的绘图程序（例如，文本编辑器、版面设置程序、电路设计程序等），用自己设计的拾取计算代替 OpenGL 的拾取机制可能会更加简单。一般很容易确定二维物体的边界框，并用分层数据结构来组织这些物体以加速搜索过程。

另外，由于只有几何物体才能引起选中事件，因此对于文本的拾取就需要程序员自己创建一种机制。设置当前的光栅位置是一种几何操作，但是它只在当前光栅位置才能有效地创建可拾取的点，通常是在文本的左下角。如果文本编辑器需要在一个文本字符串中对不同的单个字符进行操作，则必须利用其他的拾取机制。可以在拾取模式下，在每个字符周围画出一些小矩形，但是将文本作为一种特殊的情况来处理，可能会更简单。

如果决定利用 OpenGL 的拾取机制，那么程序员需要对自己的程序和数据结构进行组织，以便无论在选择模式还是在正常的绘图模式下，都可以很容易地画出合适的物体。通过这种办法，对拾取可以使用类似于屏幕绘图所用的数据结构进行操作。同样，如果程序员允许用户选中多个物体，一个办法就是用一位的信息来表示该物体是否被选中，但这种方法需要遍历所有的项目列表。为每个选中项目保持一位信息非常有用，因为在绘制整幅场景时，用户可能希望被选中的项目看起来和其他项目应有所不同（例如，将选中的项目用一个选择框包围起来）。最后，还要考虑选择用户界面。用户可能进行下面的操作：

- 选择一个项目。

- 选择一组项目。
- 在选择项中增加一项。
- 在当前选择中增加多个项目。
- 从一个选择中选定一个项目。
- 从一组重叠的项目中选出一个单项。

二维绘图程序的典型方案如下：

1）所有的选择通过鼠标光标和鼠标左键来完成。

2）点中一个项目时，该项目被选中，同时取消其他已选项。如果光标在多个项目上面，最小的一个项目被选中。

3）在没有物体的地方按下鼠标，按住并拖动鼠标光标，然后松开鼠标，这样就选中了矩形区域中所有的对象，该矩形的顶点由鼠标键按下和松开时的光标位置确定。这称为多选。所有不在扫描区域内的项目将取消选定（是完全处在扫描区域内才被选中，还是只要部分落入该区域就被选中，这一点必须加以确定。完全处在区域中才被选定的方法常常会起到较好的效果）。

4）如果用户按下 Shift 键并单击当前未选定的项，该项就会加入选择清单中。如果单击已经选中的项，则此项将从选择清单中删除。

5）如果按下 Shift 键并进行多选，那么所有扫过的项就被加到当前选中清单中。

6）在相当拥挤的区域内，常常很难进行多选。当鼠标键按下时，光标可能位于某项之上，自然该项被选中。程序员可以将每次操作都作为多选，但通常的用户界面认为，单击一项并移动鼠标，才构成选择拖动操作。

7）在多选过程中如果按下 Shift 键，在扫描区的项将加到当前的选中清单中。

8）最后，如果用户单击多个项，则只选中其中之一。如果光标不移动，用户在同一个地方再次单击，将取消上次的选择。

在特定情况下可以应用不同的规则。在文本编辑器中，不必担心字符之间的相互重叠，选定的常常是文档中连续的多个字符。因此，只要标明第一个和最后一个选定的字符，就可以确定整个选择。对文本，处理选择的最好方法就是识别字符之间的位置而不是那些字符本身。

在三维编辑器中，可对所选对象进行旋转和缩放。从另一方面来说，三维情况下的选择是比较困难的，因为在屏幕上的光标位置常常没有深度指示。

13.2　反馈

反馈与选择的相似之处在于一旦处在此模式下时，不产生任何像素，且屏幕被冻结起来。有关要渲染的图元的信息将被送回应用程序，而不是用于绘图。选择和反馈模式的最大不同之处在于发送回的是什么信息。在选择模式下，所分配的名称将返回给一整型数组。在反馈

模式下，变换过的图元信息被返回到浮点型数组中。送到反馈数组中的值，包括所处理和变换的图元类型（点、线、多边形、图像或是位图）的标志，其后为图元顶点、颜色或其他数据。返回的数值已由光照及视图操作进行了完全的变换。参数为 GL_FEEDBACK 的函数 glRenderMode()用于初始化反馈模式。

进入和退出反馈模式的基本步骤如下：

1）调用函数 glFeedbackBuffer()指定保存反馈信息的数组。此命令的参数描述了数据为何种类型，以及有多少数据写入了数组。

2）以参数 GL_FEEDBACK 调用函数 glRenderMode()，进入反馈模式（glRenderMode()的返回可以忽略）。在退出反馈模式之前，图元并不会被光栅化为像素，帧缓存的内容也不发生变化。

3）绘制图元。在绘图命令发出时，可以多次调用 glPassThrough()函数，将标记插入返回数据中，使对数据的分析变得更容易。

4）以参数 GL_RENDER 调用函数 glRenderMode()，退出反馈模式。glRenderMode()的返回值是保存在反馈数组中数值的个数。

5）分析反馈数组中的数据。

---

void glFeedbackBuffer(GLsizei size, GLenum type, GLfloat *buffer);

建立反馈数据缓冲区：缓冲区是指向存储数据的数组指针。参数 size 表示数组的大小。参数 type 描述了反馈数组中每个顶点的反馈信息，其可能值意义如表 13-1 所示。glFeedbackBuffer()必须在进入反馈模式之前调用。在表中，颜色索引模式下 k 值为 1，在 RGBA 模式下 k 值为 4。

表 13-1 函数 glFeedbackBuffer()的 type 参数的值

参数 type	坐标	颜色	纹理	值
GL_2D	x, y	-	-	2
GL_3D	x, y, z	-	-	3
GL_3D_COLOR	x, y, z	k	-	3 + k
GL_3D_COLOR_TEXTURE	x, y, z	k	4	7 + k
GL_4D_COLOR_TEXTURE	x, y, z, w	k	4	8 + k

13.2.1 反馈数组

在反馈模式下，每个要光栅化的图元要产生一批数据，并复制到反馈数组中。数据的个数由 glFeedbackBuffer()的参数 type 决定，如表 13-1 中所列。要绘制图元应使用适当的 type

值：GL_2D 或 GL_3D 用于绘制没有光照的二维或三维图元，GL_3D_COLOR 用于绘制加光照的三维物体，GL_3D_COLOR_TEXTURE 及 GL_4D_COLOR_TEXTURE 则用于绘制有光照和纹理的三维或四维图元。

每一组反馈数据都以表示图元类型的代码开始，然后是描述图元顶点和相关数据的值。另外，已经明确创建的、贯穿整个过程的标志符也由此数组返回。表 13-2 显示了反馈数组的语法结构；记住与每个返回顶点相关的数据与表 13-1 中描述一样。注意，一个多边形有 n 个返回顶点。另外，由反馈返回的 x, y, z 坐标是窗口坐标；如果返回了 w，则是裁剪坐标。对于位图和像素区域，返回的是当前光栅位置坐标。注意只有为线段重置点划模式时才返回 GL_LINE_RESET_TOKEN。

表 13-2 反馈数组的语法规则

图元类型	代码	相关数据
Point	GL_POINT_TOKEN	vertex
Line	GL_LINE_TOKEN 或 GL_LINE_RESET_TOKEN	vertex vertex
Polygon	GL_POLYGON_TOKEN	n vertex vertex ... vertex
Bitmap	GL_BITMAP_TOKEN	Vertex
Pixel Rectangle	GL_DRAW_PIXEL_TOKEN 或 GL_COPY_PIXEL_TOKEN	Vertex
Pass-through	GL_PASS_THROUGH_TOKE N	一个浮点数

13.2.2 在反馈模式下使用标记

反馈是在变换、光照、多边形拣选和由 glPolygonMode()进行多边形解释之后进行的。也可以发生在多于三条边的多边形被分割为三角形时。因此，在接收到的反馈数据中识别出所画的图元将是十分困难的。为了帮助自己分析反馈数据，在需要时应调用 glPassThrough()函数在绘图命令中插入一些标记。例如，可以用这些标记将不同图元的返回值分开。该命令将 GL_PASS_THROUGH_TOKEN 写入反馈数组中，其后是作为参数传递的一个浮点数。

---

void glPassThrough(GLfloat token);

如果在反馈模式下调用该命令，则在写入反馈数组的数据流中插入一个标记。此标记由代码 GL_PASS_THROUGH_TOKEN 及其后的一个浮点数 token 组成。在反馈模式外调用该函数不起任何作用；在 glBegin()和 glEnd()之间调用 glPassThrough()则会产生一个 GL_INVALID_OPERATION 错误。

13.2.3 一个反馈的例子

例 13-7 演示了反馈模式的用法。该程序在正常渲染模式下绘制一个有光照三维的场景。然后，进入反馈模式重画场景。由于程序绘制了有光照而无纹理的三维物体，所以反馈数据的类型为 GL_3D_COLOR。由于使用了 RGBA 模式，因此每一个未经裁剪的顶点为反馈缓冲区生成了七个数值: x, y, z, r, g, b 和 a。

在反馈模式下，程序绘制了两条直线，然后插入了一个标记。接下来，在（-100.0，-100.0，-100.0）处绘制了一个点，由于落在了正交视图体之外，因此并不在反馈数组中写入任何值。最后，插入另一个标记，并画出另一个点。

例 13-7 使用反馈模式

```
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <stdio.h>

void myinit(void);
void drawGeometry (GLenum mode);
void print3DcolorVertex (GLint size, GLint *count, GLfloat *buffer);
void printBuffer(GLint size, GLfloat *buffer);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

void myinit(void)
{
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
}

void drawGeometry (GLenum mode)
{
   glBegin (GL_LINE_STRIP);
   glNormal3f (0.0, 0.0, 1.0);
   glVertex3f (30.0, 30.0, 0.0);
```

```
    glVertex3f (50.0, 60.0, 0.0);
    glVertex3f (70.0, 40.0, 0.0);
    glEnd ();
    if (mode == GL_FEEDBACK)
    glPassThrough (1.0);
    glBegin (GL_POINTS);
    glVertex3f (-100.0, -100.0, -100.0);
    glEnd ();
    if (mode == GL_FEEDBACK)
    glPassThrough (2.0);
    glBegin (GL_POINTS);
    glNormal3f (0.0, 0.0, 1.0);
    glVertex3f (50.0, 50.0, 0.0);
    glEnd ();
}

void print3DcolorVertex (GLint size,
    GLint *count, GLfloat *buffer)
{
    int i;

    for (i = 0; i < 7; i++) {
    *count = *count - 1;
    }
}

void printBuffer(GLint size, GLfloat *buffer)
{
    GLint count;
    GLfloat token;

    count = size;
    while (count) {
    token = buffer[size-count]; count--;
    if (token == GL_PASS_THROUGH_TOKEN) {
```

```
            count--;
        }
        else if (token == GL_POINT_TOKEN) {
            print3DcolorVertex (size, &count, buffer);
        }
        else if (token == GL_LINE_TOKEN) {
            print3DcolorVertex (size, &count, buffer);
            print3DcolorVertex (size, &count, buffer);
        }
        else if (token == GL_LINE_RESET_TOKEN) {
            print3DcolorVertex (size, &count, buffer);
            print3DcolorVertex (size, &count, buffer);
        }
        }
}

void CALLBACK display(void)
{
    GLfloat feedBuffer[1024];
    GLint size;

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho (0.0, 100.0, 0.0, 100.0, 0.0, 1.0);

    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    drawGeometry (GL_RENDER);

    glFeedbackBuffer (1024, GL_3D_COLOR, feedBuffer);
    (void) glRenderMode (GL_FEEDBACK);
    drawGeometry (GL_FEEDBACK);

    size = glRenderMode (GL_RENDER);
    printBuffer (size, feedBuffer);
```

```
}


int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("编程指南例 13-7");
    myinit ();
    auxMainLoop(display);
    return(0);
}
```

运行结果如图 13-4 所示。

图 13-4 反馈模式实例

因此，下面的命令绘制出的线条将产生两个图元。

```
glBegin(GL_LINE_STRIP);
    glNormal3f (0.0, 0.0, 1.0);
    glVertex3f (30.0, 30.0, 0.0);
    glVertex3f (50.0, 60.0, 0.0);
    glVertex3f (70.0, 40.0, 0.0);
glEnd();
```

第一个图元以 GL_LINE_RESET_TOKEN 开始，表示此图元是一条线段并重设了线的点划模式。第二个图元也以 GL_LINE_TOKEN 开始，因此也是一条线段，但是线的点划模式并没有重设。注意，这两条线段四个顶点的 RGBA 值为（0.84，0.84，0.84，1.0），这是一种 alpha 值为最大的浅灰色。这些颜色是表面法线和光照参数相互作用的结果。

由于在两个标记之间没有产生反馈数据，所以可以得到结论，在最初的两次glPassThrough()调用之间绘制的任何图元，都从视图体中裁剪掉了。最后，在（50.0，50.0，0.0）处的点被画出来，相关的数据也被复制到反馈数组中。

试一试

- 修改例 13-7，看看这些改变如何影响返回的反馈值。例如，可以改变 glOrtho()的坐标值；改变光照变量，或者去掉光照并将反馈类型改为 GL_3D。

第 14 章　OpenGL 应用及提高应用程序性能的技巧

本章主题

OpenGL 是一种底层的工具包；在前面的章节中读者已经了解了这些工具的用法，可以实现高层的功能。本章给出了几个实现这样高层功能的例子。

本章还讨论了基于 OpenGL 命令的几项技术，讲解了关于这些命令的几项不太容易想到的用法。给出的例子并不彼此相关，也没什么特别的顺序。建议阅读本节开始的介绍后，可以直接找到感兴趣的内容。为了方便后面的阅读，在这里列出了这些例子并做了简要的说明。

注意，本书的大部分例程都是完整的，可以直接编译执行。但本章例程的代码并不完整，如果想运行，需要进行进一步的工作。

本章介绍的主要内容如下：

- “轻松制作透明物体”，解释了如何使用多边形点划得到半透明物体；在没有混色硬件时这一技术尤其有用。
- “轻松实现淡出效果”，演示了如何使用多边形点划来创建物体淡出效果（一种消隐过程，最后变为背景颜色）。
- “利用后缓存进行物体选择”，描述了如何在一个双缓存系统中用后缓存处理简单的物体选择。
- “方便的图像变换”，讨论了如何通过将每个像素画成四边形，绘制一幅位图化的图像的扭曲的形式。
- “分层显示”，解释了如何显示材质的多个不同层次并指明材质在何处进行重叠。
- “字符反走样”，描述了如何绘制光滑的字体。
- “绘制圆形的点”，描述了如何画近似为圆的点。
- “插入图像”，展示了如何平滑地把一幅图像混入到另一幅图像。
- “制作贴花”，解释了如何绘制两幅图像，其中一幅作为贴花总在另一幅之上。
- “用模板缓存绘制填充的凹多边形”，解释如何利用模板缓存绘制凹多边形、复杂多边形和带孔洞的多边形。
- “确定重叠区域”，描述了如何确定三维空间中的重叠区域。
- “阴影”，描述如何绘制光照物体的阴影。
- “消除隐藏线”，讨论如何利用模板缓存删除线框物体隐藏掉的线。
- “纹理映射的应用”，描述了几个纹理映射的技巧，比如图像的旋转和扭曲。
- “绘制深度缓存图像”，讲解如何在深度缓存环境内结合入图像。

- “笛里希累(Dirichlet)域”，解释如何确定使用了深度缓存的一组点中的 Dirichlet 域。
- “生命游戏（Life Game)”，解释如何用模板缓存实现动画游戏。
- “交替使用 glDrawPixels()和 glCopyPixels()”，描述了如何在一些特殊效果，比如仿视频、油漆喷雾和图像反转中使用这两个命令。

14.1 轻松制作透明物体

可以利用多边形点划来模拟透明材质。对于缺少混色硬件的系统来说，这是一种很好的办法。由于多边形点划方式是 32×32=1024 位的，从不透明到透明的过渡过程共包括 1023 步。例如，如果希望表面透光为 29%，只需要简单地编辑点划模式为：将 29%（大约 297）的屏蔽像素点置为 0，其余置为 1。注意，即使各表面具有相同的透明度，也不要对各表面使用同一种点划模式，因为这些表面在屏幕上覆盖的位完全相同。通过随机选定恰当数目的像素点为 0 来编辑不同的模式。

如果对随机像素点所出现的透光效果不满意，可以使用矩形模式，但当各透明表面出现重叠时，使用该模式所达到的效果也不太好。由于在大多数的场景中几乎没有透明区域重叠的情况，所以这一点倒是不成问题。在带透明窗口的汽车图片中，人的视线最多能穿过两块玻璃。

14.2 轻松实现淡出效果

如果要得到一幅渐渐消隐到某种背景色中的图像，首先定义一系列的多边形点划模式，每一种都有更多的位得以显示，这样来代表越来越浓的形式。然后，在一个足够大的多边形中重复使用这些模式，该多边形至少要覆盖整个消隐区域。例如，假设希望用 16 步就消隐到黑色。首先定义 16 个不同的模式数组：GLubyte stips[16][4*32]; 然后载入它们，使得在 32×32 的显示的点划模式中，每种占 1/16 的像素数。下面的代码实现了以上过程：

```
draw_the_picture();
glColor3f(0.0, 0.0, 0.0);    /* 将颜色设置为黑色 */
for (i = 0; i < 16; i++)
    glPolygonStipple(&stips[i][0]);
    draw_a_polygon_large_enough_to_cover_the_whole_region();
}
```

在有些 OpenGL 实现当中，先将点划模式编译成显示列表会得到更好的性能。在初始化的过程中，加入如下代码：

```
#define STIP_OFFSET 100
for (i = 0; i < 16; i++)
```

```
    glNewList(i+STIP_OFFSET, GL_COMPILE);
    glPolygonStipple(&stips[i][0]);
    glEndList();
}
```

然后，将在第一个代码段中的 glPolygonStipple(&stips[i][0])用 glCallList(i)代替。

通过编译该命令将点划设置为显示列表，OpenGL 可以将数组 stips[ ][ ]中的数据重排列成硬件为了获得最大的点划设置速度要求的格式。

这项技术的另一个应用是在绘制变化图像时，希望得到图像运动之后留下渐渐消隐的动态模糊效果。例如模拟行星系统，希望能够在运动行星的后面留下尾迹以显示行星的轨迹。仍然假设用 16 步完成消隐，设置与前面一样的点划模式（比如用显示列表的形式），程序中主循环设置如下：

```
current_stipple = 0;
while (1) {                        /* 设置成死循环 */
    draw_the_next_frame();
    glCallList(current_stipple++);
    if (current_stipple == 16) current_stipple = 0;
    glColor3f(0.0, 0.0, 0.0);       /* 设置颜色为黑色*/
    draw_a_polygon_large_enough_to_cover_the_whole_region();
}
```

每次循环，清除 1/16 的像素点。在 16 帧之内都没有行星的像素点全部设置为黑色。当然，如果系统支持硬件混色，将很容易实现每一帧与一定量的背景颜色混合。

14.3　利用后缓存进行物体选择

尽管 OpenGL 的选定机制功能强大而且使用灵活，但用起来却很麻烦。通常的情况比较简单：应用程序画了一个场景，由一些实体组成；用户用鼠标点某个物体时，应用程序就找出鼠标尖端指向的物体。

要做到这一点，应用程序必须使用双缓存模式。当用户选定一个物体时，应用程序在后缓存中重绘制整个场景，但并不使用物体本身的颜色，而是将每种物体的颜色编码为某种物体的标识符。应用程序只需要读回光标下的像素点，该像素点的值就对点击物体的数目进行编码。

记住，如果在同一像素点有多个物体相互重叠，这种机制将在标准选定中优选前面的物体。由于用假颜色绘制的图像画在后缓存中，所以对用户而言是不可见的；程序员可以在交换缓存之前重画后缓存（或将其从前缓存中拷贝出来）。在颜色索引模式，编码过程只是将物体的标识作为索引来发送。在 RGBA 模式，将标识符的位编码到 R，G，B 各分量中。

记住，如果在场景中有多个物体，标识数可能溢出。例如，假设你在每一颜色缓存使用 4 位颜色索引信息的系统中使用颜色索引（即共有 16 种可能的索引值），而整个场景却有上千个可选项。为了解决这个问题，选定可分几次扫描完成。为了明确起见，假设场景中物体数少于 4096 个，这样用 12 位就可以对所有的物体编码。第一次扫描，用高 4 位索引画场景，然后用中间 4 位和低 4 位来画。在每一次扫描之后，读取光标下的像素点，提取各位，最后，将它们集合到一起得到物体标识数。

用这种方法，共要进行三次选定，但这通常是可以接受的。记住，在得到高 4 位之后，排除 15/16 种可能，所以第二次仅需要画 1/16。类似地，第二次之后，排除了 255/256 的可能物体。第一次扫描需要画一个框架的时间，但第二次和第三次速度可以达到第一次的 16 倍和 256 倍。

如果希望写出能够在各不同的系统之间可以移植的代码，将物体标识符分成块以适应这些系统的最小公分母。记住，系统在 RGB 模式下可能自动执行抖动。如果是这样，关掉抖动。

14.4 方便的图像变换

假设想将一幅位图化的图像扭曲（可以是简单地拉伸或旋转，或用数学函数进行复杂地变换）。在许多情况下，通过将每一像素点的图像画成四边形就可以实现令人较为满意的效果。尽管这种方法的效果不如应用复杂的过滤算法（而且，对于资深程序员也是不够的），但它十分快捷。

更具体描述这个问题，设原始的图像是 m×n 像素的，其坐标选自范围[0, m-1]×[0, n-1]中。设变形函数为 x(m,n)和 y(m,n)。例如，如果变形仅是简单地按照 3.2 的因子进行缩放，变形函数就设为：x(m,n) = 3.2×m 和 y(m,n) = 3.2×n。下面的代码绘制了变形后的图像。

```
glShadeModel(GL_FLAT);
glScale(3.2, 3.2, 1.0);
for (j=0; j < n; j++)
glBegin(GL_QUAD_STRIP);
for (i=0; i <= m; i++)
glVertex2i(i,j);
glVertex2i(i, j+1);
set_color(i,j);
}
glEnd();
}
```

这段代码使用单色画每个变换后的像素点，颜色与该像素点相同，并将图像尺寸放大 3.2

倍。函数语句 set_color()代表设置图像像素颜色的任意 OpenGL 命令。

下面代码稍加复杂，利用函数 x(i,j) 和 y(i,j)进行图像变形。

```
glShadeModel(GL_FLAT);
for (j=0; j < n; j++) {
   glBegin(GL_QUAD_STRIP);
   for (i=0; i <= m; i++) {
      glVertex2i(x(i,j), y(i,j));
      glVertex2i(x(i,j+1), y(i,j+1));
      set_color(i,j);
   }
   glEnd();
}
```

用如下代码可得到更好的变形效果。

```
glShadeModel(GL_SMOOTH);
for (j=0; j < (n-1); j++)
glBegin(GL_QUAD_STRIP);
for (i=0; i < m; i++)
set_color(i,j);
      glVertex2i(x(i,j), y(i,j));
      set_color(i,j+1);
      glVertex2i(x(i,j+1), y(i,j+1));
   }
   glEnd();
}
```

这段代码沿四边形对颜色进行平滑插值。记住，比起无明暗效果的阴影形式，这种形式在每一个方向上少一个四边形，因为使用了颜色图像来指定四边形顶点的颜色。另外，可以通过合适的混合函数（GL_SRC_ALPHA, GL_ONE）来消除多边形的锯齿现象以得到效果更好的图像。

14.5　分层显示

在有些应用中，比如半导体设计程序，希望能够显示不同材质层次，并指明不同材质重叠的位置。

举一个简单的例子，假设你有可分层的三种不同的材质。每一点有 8 种不同层次的组合，如表 14-1 所示。

表 14-1 8 种不同层次组合

	层 1	层 2	层 3	颜色
0	absent	absent	absent	黑
1	present	absent	absent	红
2	absent	present	absent	绿
3	present	present	absent	蓝
4	absent	absent	present	粉红
5	present	absent	present	黄
6	absent	present	present	白
7	present	present	present	灰

这里希望程序能显示 8 种颜色，每层各不相同。在表的最后一列给出任意的颜色。为了达到这种效果，必须将颜色模式设置为索引模式并进行映射，使得第 0 条为黑，第 1 条为红，第 2 条为绿，等等。

为了清除窗口，将写屏蔽设为 7（对所有的 3 层）并设清除颜色为 0。为了绘制图像，将颜色设为 7，之后想在第几层绘制一些物体，就将写屏蔽设为几。在别的类型的应用中，可能需要在某一层中有选择地擦除，这时可使用与前面一样的写屏蔽，但需要将写屏蔽颜色设为 0 而不是 7。

14.6 字符反走样

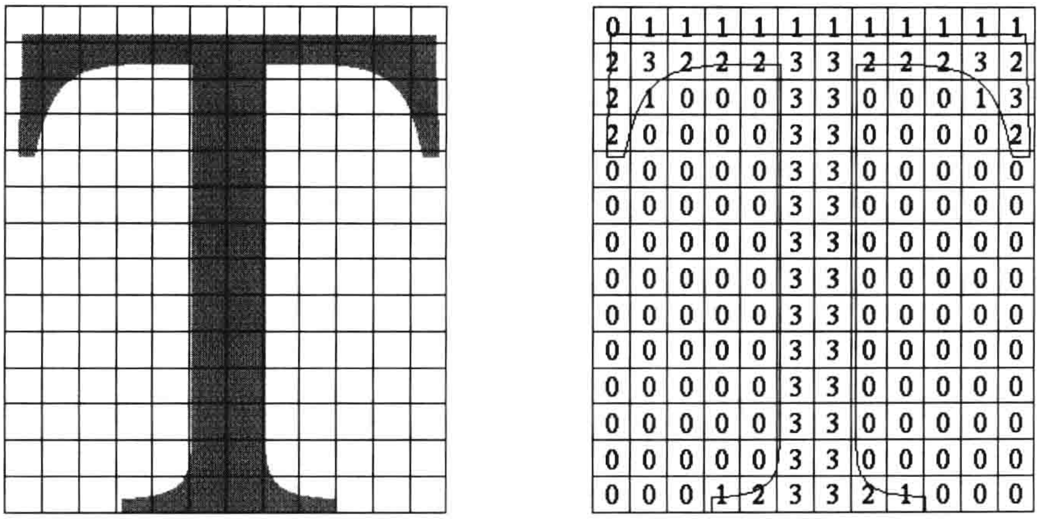
标准的字符绘制技术是用函数 glBitmap()，所画的字符像素是黑白分明的。例如在黑色的背景上写白色的字符，结果像素点要么为黑要么为白，绝不会是灰色的明暗效果。如果在渲染字符时采用中间颜色（在这个例子中，可采用灰色），可以获得更为平滑和高质量的图像。

若在白色背景上绘制黑色字符，想象一个屏幕上像素点的高度放大图像，并在其上添加高分辨率的字符轮廓，如图 14-1（a）所示。

图中可以注意到有些像素点完全被字符轮廓所覆盖，应画为黑色；完全处在轮廓之外的像素点则应画成白色；而有些像素点划成灰色的明暗效果最理想，灰色中的黑色成分对应着像素的黑色量。使用这项技术将进一步改善图像的效果。

如果不考虑速度和内存量，每一个字符就可以画成小图像而不画成位图。然而如果使用 RGBA 模式，这种方法需要为字符的每个像素存储和重画多达 32 位，而不是像普通的字符那样每一像素只需要一位。另外，还可以使用每个像素 8 位的索引，并在转换的过程中通过查表将这些索引转换为 RGBA 模式。在多数情况下，使用黑与白之间几个不同层次的灰色

（比方说，2 个或 3 个）来作为过渡，得到的字体每位像素仅需要 2 或 3 位的存储。



（a）　　（b）

图 14-1　字符反走样

图 14-1（b）中的数字指明了每个像素点所覆盖的估计百分比：0 意味着几乎为空，1 意味着约覆盖 1/3；2 为 2/3；3 是完全覆盖。如果标记为 0 的像素画成白色，标记为 3 的画成黑色，标记为 1 和 2 的分别画成 1/3 黑和 2/3 黑，这样一来，得到的字符就有很好的效果。存储 0，1，2，3 四个标记只需要 2 位，这样每一像素点 2 位就可以存储 4 个灰度层次。

一般情况下有两种方法实现字符反走样，这取决于是否处于 RGBA 模式下。

在 RGBA 模式下，定义三种不同的字符位图，相应于在图 14-1 中的 1，2，3。将背景颜色清为白色。将颜色设为 1/3 灰度（RGB = (0.666, 0.666, 0.666)）并将所有的像素点以标记 1 来画。然后设置 RGB = (0.333, 0.333, 0.333)，用来画位图 2，设置 RGB = (0.0, 0.0, 0.0) 画位图 3。这样定义三种不同字体，将字符串重画三次，每次扫描填充适当的颜色度的位。

在颜色索引模式下也类似，但如果希望正确设置颜色图映射并使用写屏蔽，可以通过每个字符仅两个位图、每个字串扫描两次得以脱离。在前面的例子当中，设置有一个 1 的位图，无论在该字符处出现的为 1 还是 3。同样无论出现的为 2 或 3，均将第二个位图设为一个 1。载入颜色映射图，0 表示白色，1 代表浅灰色，2 代表深灰色，3 表示黑色。将颜色设置为 3（用二进制表示为 11），而写屏蔽为 1，绘制第一个位图。然后，将写屏蔽设为 2，绘制第二个位图。在图 14-1 中 0 出现的地方，帧缓存中进行的是空操作。在 1，2，3 处，帧缓存中也出现 1，2，3。

对于有 4 个灰色层次的本例而言，存储量减小了——只需要两次扫描而不是三次。如果使用 8 种灰色层次，RGBA 模式就需要 7 次扫描，而颜色映射图屏蔽技术只需要 3 次。如果是 16 种灰色，则两种模式分别需要扫描 16 次和 4 次。

试一试

- 是否可以实现用不多于优化颜色索引情况的图像数来进行 RGBA 渲染？提示：考虑在需要反走样时，RGB 片原是如何融合进颜色缓存的。

14.7 绘制圆形的点

通过激活点的锯齿现象的消除、关闭混合并使用只传递 alpha 值大于 0.5 的片原的 alpha 函数，可以绘制接近圆形的、消除锯齿现象的点。

关于这些主题参见“消除锯齿现象”和“混色”两节。

14.8 插入图像

假设有两幅图像（这里图像可能是位图图像或用通常的几何方式产生的图片），希望能平滑地由一幅过渡到另一幅。这用 alpha 颜色分量和合适的混色操作很容易做到。假定在 10 步之内实现混色，图像 A 在第 0 帧显示，图像 B 在第 9 帧显示。显然，在第 i 帧中，图像 A 的 alpha 设为(9-i)/9 而图像 B 设为 i/9。这种方法的问题在于，在每一帧中两幅图像都必须画。一种较快的方法是将图像 A 画在第 0 帧。将图像 B 的 1/9 与当前图像（完整的图像 A）的 8/9 混合得到第 1 帧。图像 B 的 1/8 与当前图像的 7/8 混合得到第 2 帧。对第 3 帧为图像 B 的 1/7 与当前图像的 6/7 相混合。最后一步，将图像 B 的 1/1 与剩下图像的 0/1 相混合，恰好得到图像 B。对于第 i 帧，将图像 B 的 1/(9-i)与当前处的(8-i)/(9-i)混合。

14.9 制作贴花

利用深度缓存绘制清除了掩藏表面的复杂三维图形时，进一步假设图的一部分是图 A 和 B 的公共面，其中 B 作为贴花总在图 A 上。

第一种方法是必须画完 A 之后再画 B，将 B 的深度缓存函数设置为等于或大于 A。由于代表顶点的浮点数的精度是有限的，取整的精度误差常引起多边形 B 在 A 之前一点或之后一点。下面是这个问题的一种解决方法的步骤：

1）取消深度缓存的写入状态，然后渲染 A。

2）激活深度缓存的写入状态，然后渲染 B。

3）取消颜色缓存的写入状态，再次渲染 A。

4）激活颜色缓存的写入状态。

注意，在整个过程中，要激活深度缓存的检测。在第一步，无论 A 在什么位置都要渲染，但深度缓存的值不改变，同样在第二步，无论 B 在 A 上何处显示，都要保证将 B 画出。第三步只保证在 A 下所有深度值要进行正确更新，但由于 RGBA 的写失效，像素点的颜色

值不受影响。最后一步返回系统的缺省状态（即，深度和颜色缓存均可写）。

如果能够使用模板缓存，可用下面更简单的技术：

1）如果深度检测通过，设定模板缓存写 1，否则为 0。渲染 A。

2）设定模板缓存的模板值不改变，而只渲染模板值为 1 的地方。取消使深度缓存检测及其更新。渲染 B。

用这种方法，任何时候都不必将模板缓存的内容初始化，因为在渲染 A 时所有感兴趣的像素模板值（即，那些由 A 渲染的像素）都将得以设置。在绘制其他多边形之前要保证再次激活深度检测而取消模板检测。

14.10 用模板缓存绘制填充的凹多边形

考虑图 14-2 所示的凹多边形 1234567。假设此多边形是以一系列三角形画出的：123，134，145，156，167，如图 14-2 所示。粗线代表初始多边形边界。画出所有这些矩形将缓存划分为九个区域：A，B，C，.. I，区域 I 在所有的三角形之外。

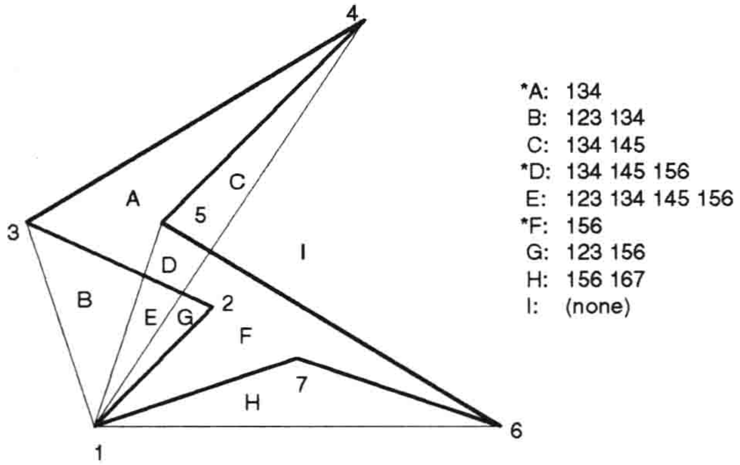


图 14-2 凹多边形

图中的文字区列出了每个区域的命名和覆盖它的三角形。区域 A，D 和 F 由原多边形组成；注意这三个区域由奇数的三角形覆盖。其他的区域由偶数的三角形（可能为 0）覆盖。这样，渲染凹多边形的内部时，只需要渲染奇数的区域。这可以使用双扫描算法通过模板缓存来完成，

首先，清除模板缓存并取消颜色缓存的写入状态。下一步，依次绘制每一个三角形，这通过在模板缓存中使用 GL_INVERT 函数实现。在每绘制一次三角形覆盖一个像素时，其值将在 0 与非 0 之间跳变。当所有的三角形画完时，模板缓存的值为 0；否则为非 0。最后，绘制一个大多边形覆盖整个区域（或重画三角形），但只允许在模板缓存为非 0 的区域进行绘制。

注意：前述方法有一些普遍性，并不需要必须从多边形的顶点开始。在 1234567 的例子

中，P 可在多边形的内或外。绘制三角形：P12，P23，P34，P45，P56，P67 和 P71。奇数三角形所覆盖的区域在内部；其他的区域在外部。其普遍性在于，即使 P 恰巧位于多边形的边上，一个三角形为空即可。

这项技术既可以应用于简单多边形（多边形各边相互连接）也可应用于孔洞多边形。下面的例子说明了如何处理带两块区域的复杂多边形：一块是四边的，另一块是五边的。进一步假设有一个三角形和一个四边形两个孔洞（位于哪一个区域内无所谓）。设这两个区域为 abcd 和 efghi，孔洞为 jkl 和 mnop。设 z 为平面内的任一点。绘制如下的三角形：

zab zbc zcd zda zef zfg zgh zhi zie zjk zkl zlj zmn zno zop zpm

奇数的三角形覆盖的区域标志为 in，偶数的标志为 out。

14.11 确定重叠区域

在设计一个小的三维块构成的各个机械部分时，常希望显示出各部分的交叠区域。许多情况下，这样的区域能指示设计错误：机械设计部件之间发生冲突。对于运动部件的情况，这种显示更有价值，因为冲突部分的查找将贯穿整个机械设计周期。实现这一技术的方法很复杂，不是三言两语能够穷尽的。详细的介绍可查阅相关文献。

这种方法与“模板检测”中描述的截顶算法相关。基本的想法是设置一个穿过需要检测重叠的物体的裁剪平面，然后确定何时一部分裁剪平面一次切割入不止一个物体内部。对于静态图像，裁剪面可手工移动到冲突区的加亮部分；对于动态图像，使用栅格裁剪面来查找可能的重叠区将更简单。

绘制想要检测的物体，然后按照裁剪平面来裁剪。注意确定哪些在裁剪平面上的像素点在物体内部用模板缓存中的奇偶来计算，这在前一部分已经解释（对于规则物体，如果从点到眼睛的连线穿过物体的奇数个面，那么该点就在物体的内部）。要找到冲突区，需要找出这样的帧缓存中的像素点：此时的裁剪平面同时在两个或两个以上的物体内部；换句话说，在任何两个相交物体的内部。

对每一个新物体，要确定它的内部，找出其与至今检测的物体内部并集的交集，并跟踪叠合点。然后将新物体的内部点加到其他物体内部并集中。

程序员可以通过同时使用模板缓存中的不同位和各种屏蔽操作来实现前面描述的操作。每个像素需要三位模板缓存——一位确定每一物体的内部；一位记录当前发现的所有内部的并集；一位记录目前发生重叠的区域。说得更具体些：假设模板缓存的位 1 是为了确定内部/外部，位 2 是在增加的内部区域并集，位 3 是至今发现的重叠区域。对于将要渲染的物体，清除位 1（使用模板屏蔽 1 并清为 0），然后使用 GL_INVERT 模板操作并保持模板屏蔽为 1，以触发位 1。

用模板操作可以找到在模板缓存中的各位的冲突和并集。例如，要把缓存 2 的位作为缓存 1 和 2 中的位的并集，需将模板屏蔽为那两位。如果缓存 1、缓存 2（或两个一起）中的

位在线，就会出现上述情况。如果对照成功，在缓存 2 中写入一个 1。同样，此时要保证颜色缓存中的图处在取消状态。交集的计算是类似的——只在两个缓存中的值为 3 时（缓存 1，2 中的位都在线）才使函数通过。将结果写入正确的缓存。

14.12　阴影

每一可能的从三维空间到三维空间的投影，均可用合适的 4×4 可逆矩阵和齐次坐标得到。如果矩阵不可逆而且其秩为 3，将把三维空间投影到二维平面。每一这样的投影均可由适当的秩为 3 的 4×4 矩阵得到。要得到任意平面上的任意物体在任意光源下的阴影（可能在无穷远处），需要得到代表该投影的矩阵，在矩阵堆栈中相乘，并用阴影颜色画物体。需要记住的是，需要投影到背景的各平面。

作为简单的说明，假设光源在原点，背景平面的方程为 ax+by+c+d=0。如果给定顶点 S=(sx,sy,sz,1)，穿过 S 的光线包括所有的点 aS，其中 a 是任意的实数。线与面的交点在：

a(a×sz+b×sy+c×sz) + d = 0

所以

a = -d/(a×sx+b×sy+c×sz).

将其代回原方程，得到交叉点为：

-d(sx,sy,sz)/(a×sx+b×sy+c×sz)

第一次变换场景时，光源置于原点，可以使用该矩阵。如果光源在无限远处，所知道的仅是点 S 和方向 D = (dx,dy,dz)。直线上的点由下式给出：

$S + \alpha D$

如前所述，光线与平面的交点由下式给出：

$a(sx+\alpha dx)+b(sy+\alpha dy)+c(sz+\alpha dz)+d = 0$

14.13　消除隐藏线

绘制一个删除了隐藏线的线框物体，一种有效的方法是用线画出轮廓，然后用带背景颜色的多边形来填充组成表面的多边形内部。激活深度缓存，填充的内部覆盖了所有的距离眼睛较远、被其他较近的面挡住了的轮廓线。只要保证物体完全落在多边形轮廓的内部，这种方法就可以使用，并且事实上物体在各种位置都可能与多边形轮廓重叠。

用模板缓存有一容易的二次扫描的解决办法。对每一多边形，清除模板缓存，然后在帧缓存和模板缓存中都进行轮廓绘制。那么填充内部时，仅在模板缓存仍为空的位置激活绘画即可。为了避免对每一多边形都清除整个模板缓存，一种简单的清空方法是，用同一多边形轮廓在缓存内画 0。此时仅需要清除整个模板缓存一次。

例如，如下是实现删除隐藏线的内循环代码。每一多边形用前景颜色画出轮廓。模板缓

存用来避免多边形的填充色覆盖其轮廓线。为了优化性能，每一循环中，通过在绘制多边形轮廓时使用同一值，只改变模板和颜色参数两次。

```
glEnable(GL_STENCIL_TEST);
glEnable(GL_DEPTH_TEST);
glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS, 0, 1);
glStencilOp(GL_INVERT, GL_INVERT, GL_INVERT);
set_color(foreground);
for (i=0; i < max; i++) {
    outline_polygon(i);
    set_color(background);
    glStencilFunc(GL_EQUAL, 0, 1);
    glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
    fill_polygon(i);
    set_color(foreground);
    glStencilFunc(GL_ALWAYS, 0, 1);
    glStencilOp(GL_INVERT, GL_INVERT, GL_INVERT);
    outline_polygon(i);
}
```

14.14 纹理映射的应用

纹理映射功能强大，有许多有趣的应用。这里给出纹理映射的几个高级应用。

文本锯齿现象消除——为每一字符定义一个相对较高分辨率的纹理映射，然后使用纹理化提供的过滤将其映射到较小的区域。即使文本处在未与屏幕排齐的表面上，看来也是正确的，但文本倾斜翘起并且有些透视变形。

线锯齿现象消除——这与文本的锯齿现象消除相似：将纹理中的线制成几个像素宽度，使用纹理过滤消除锯齿现象。

图像缩放和旋转——若将图像放入纹理映射图，并用该纹理映射到一个多边形上，旋转缩放多边形也就相当于有效地旋转和缩放了图像。

图像变形——如前述例子，将图像存储为纹理映射图，不同的是将其映射到样条曲面（使用求值程序）。该曲面变形时，图像也就跟着变形。

图像投影——将图像放入纹理映射图，并作为点光源投影，造出滑动投影效果。

关于旋转和缩放请参见第 4 章；关于创建纹理，请参见第 10 章；关于求值程序，请参见第 12 章。

14.15 绘制深度缓存图像

对于复杂的静态背景，对背景的几何描述的渲染，要比绘制背景已经渲染的像素图像用更长的时间。如背景固定而前景只简单地变化，可以将背景和相关的深度缓存作为图像的一种形式画出，而不必从几何学意义上渲染。前景也可能由需要花费时间渲染的物体项目组成，但其帧缓存和深度缓存是能够获得的。程序员可以使用二次扫描算法将这些项目渲染到深度缓存环境。

例如，若绘制一个球体的分子模型，可以得到渲染得非常漂亮的球体图像，还可以用 Phong 阴影或光线跟踪计算，或者使用其他通过 OpenGL 不能直接得到的方法，得到此球体相关的深度缓存值。要绘制一个复杂的模型，可能需要画几百个这样的球体，也就需要几百个深度缓存。

要在场景中加入深度缓存图像，首先要使用 glDrawPixels()将图像的深度缓存值画到深度缓存中。然后激活深度缓存，设置写屏蔽为 0 时没有绘画操作进行，激活模板，使得一旦发生了模板缓存写入，便画模板缓存。然后将图像画入颜色缓存，该颜色缓存由刚刚执行写入的模板缓存屏蔽，从而使得仅在模板缓存为 1 时才进行写入操作。写的过程中，要在模板缓存之外将模板化函数设为 0，这样当将下一幅图像加到场景中时进行自动清除。如果物体移近或移离观察者，需要使用正交投影；在这种情况下，要使用带有参数 GL_DEPTH_BIAS 的 glPixelTransfer*()来移动深度图像。

关于 glDrawPixels() 和 glPixelTransfer*()参见“删除隐藏面挽救工具箱”，“深度检测”，“模板检测”等节和第 9 章。

14.16 笛里希累（Dirichlet）域

给定平面上的一组点 S，某点的 Dirichlet 域或 Voronoi 多边形是指在平面上，比点集 S 中的点更靠近这一点的点集。这些点为计算几何的许多问题提供了解决方案。图 14-3 表示了一组点的 Dirichlet 域轮廓。

若绘制一个深度缓存锥体，使其顶点的颜色与 S 中各点颜色均不相同，且每一点的 Dirichlet 区域都用该点的颜色。最简单的方法是预先计算图像中的锥体深度，并用前面一节描述的方法将图像作为深度缓存值使用。这里并不需要像画阴影球体时那样将图像画到帧缓存，但将图像画到深度缓存时，需要使用模板缓存，通过先清除像素点，然后在深度测试通过以后写入非零值，用深度缓存记录进行绘图的像素。要画出 Dirichlet 区域，在整个窗口覆盖上一个多边形，但只在模板缓存非 0 时才激活绘图。

用一个简单的深度缓存并用同一种颜色渲染锥体会更简单，要达到良好的效果可能需要上千个多边形。本节给出的方法能够更快捷地渲染出更高质量的锥体。

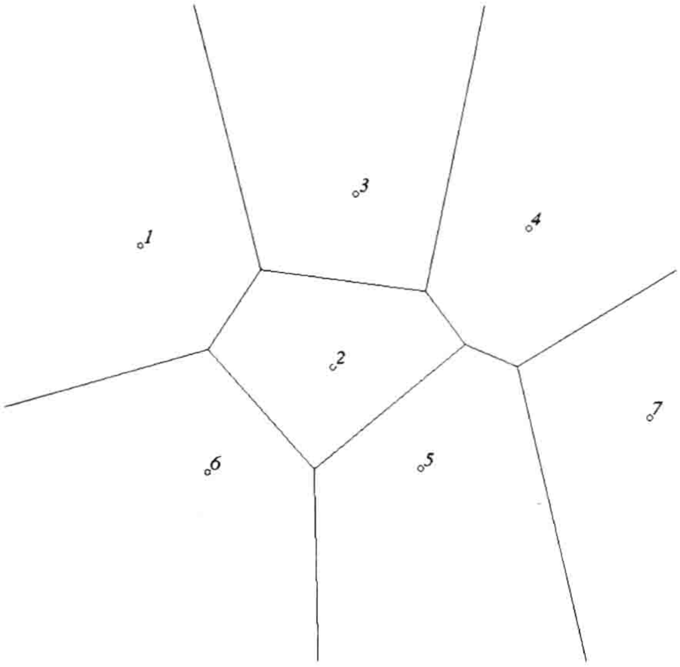


图 14-3 Dirichlet 区域

14.17 生命游戏（Life Game）

由 John Conway 发明的生命游戏，是在一个矩形栅格区内进行的，其每一栅格或“活”或“死”。要根据当前状态计算其下一状态，需要对每一个栅格位置计算其活的邻格（指每一栅格区的八个邻近格）数目。如果在第 n 代是活的而且有 2 个或 3 个活的邻格，或者在第 n 代是死的而有 3 个活的邻格，那么该栅格位置在第 n+1 代是活的。其他情况，第 n+1 代都是死的。不同的初始设置可以产生许多令人难以置信的有趣的模式。（关于数学游戏方面的内容请参见相关资料）。图 14-4 展示了生命游戏中的 6 代。

用 OpenGL 来创建游戏的一种方法是用多次扫描算法。将数据保存在颜色缓存中，每一个栅格点用一个像素。假设以黑色（全为 0）为背景色，活的像素点颜色不为 0。将深度缓存和模板缓存初始化为 0，设置深度缓存写屏蔽为 0，并设置深度比较函数使能在不等的时候进行扫描。为了重申说明，从屏幕上读取图像，激活对深度缓存的绘图，设置模板函数使其在深度比较成功时增加，否则模板缓存不变，取消对颜色缓存的绘图。

下一步，将图像画 8 次，在垂直、水平和对角线方向分别偏置一个像素。在完成时候，对每一像素，模板缓存包含其活邻格的计数。激活对颜色缓存的绘图，将其颜色设置成活单元的颜色，并当模板缓存的值为 3（3 个活的邻格）时，设置模板函数为可画。另外，当进行绘图操作时，减少模板缓存的值。然后，绘制一个矩形覆盖整个图像；用“活”单元的颜色来画有 3 个活邻格的单元。

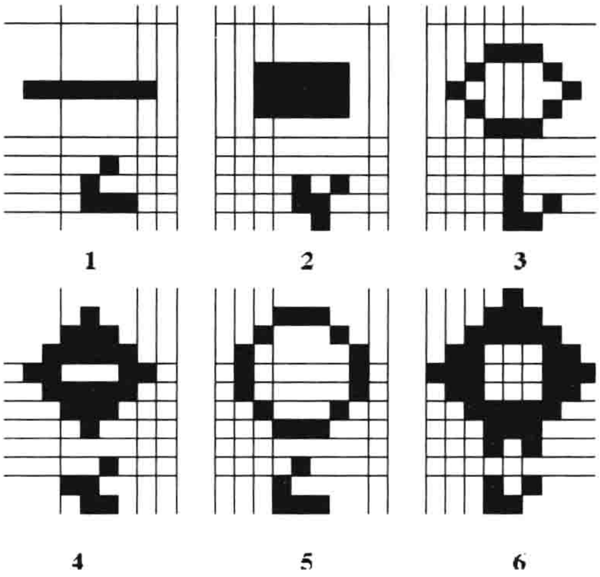


图 14-4　生命游戏中的六代生命体

在这一点，模板缓存包含 0，1，2，4，5，6，7，8，并且处在 2 下的值是正确的。处在 0，1，4，5，6，7 和 8 下的值必须清为“死”颜色。设置模板函数在值不为 2 时进行绘图，并在各种情况下设置模板值为零。然后用“死”颜色画一个大多边形覆盖整个图像。整个过程完成。

对可用的演示程序，可能需要将一个栅格放大到大于单个像素的尺寸；用每一栅格点一个像素，很难看到图案细节。

14.18　交替使用 glDrawPixels()和 glCopyPixels()

读者可能将 glDrawPixels()看成是将矩形像素区绘制到屏幕上的函数。尽管这是此函数主要的用途，但它还有一些其他的用法。

视频——即使计算机没有特殊的视频硬件，也可以使用 glDrawPixels()在后缓存的同一位置反复画每一帧，然后与前缓存交换，来实现播放简短的动画剪辑。这种方法播放时，为了显示合理的效果所需要的帧尺寸，依赖于硬件的处理速度，因此要想得到平滑的视频，可以将电影效果限制在 100×100 像素区域内。

画刷——在绘图程序中，画刷形状用 alpha 值模拟。绘画的颜色由颜色值表示。用蓝色的圆刷绘制，即相当于用 glDrawPixels()重复画一个蓝色的正方形，其 alpha 值在中间最大，逐渐过渡到以正方形中心为原点的圆的边界时为 0。使用混合函数进行绘图，该函数用引入颜色的 alpha 值和原在像素点的（1- alpha）颜色进行混合。如果画刷的 alpha 值均小于 1，就必须重复画一个区域以得到单色。如果 alpha 值接近 1，画刷的每一笔将抹去下面的颜色。

过滤放大——如果用一个非整型量放大像素图像，OpenGL 将使用一个方盒过滤器——这可能导致更严重的锯齿效应。为了改善过滤效果，将得到的图像在小于一个像素的范围内

抖动并重画多次，用 alpha 混和使得到的像素作平均。结果就是过滤放大。

图像调换——用 glCopyPixels()的 XOR 操作可以交换相同尺寸的图像的位置。用这种方法，可以避免将图像读回处理器内存中。如果 A 和 B 代表了两幅图像，操作如下：

1）A = A XOR B

2）B = A XOR B

3）A = A XOR B

第 15 章　高级应用（一）：OpenGL 与 AutoCAD 的图形接口

本章主题

通过本书前面章节的详细阐述，我们可以清晰地看到，OpenGL 这一完全开放的图形开发环境为程序员提供了强大的二维及三维图形功能，这些功能包括：建模，变换，光照，色彩，明暗处理，运动，纹理，材质处理，以及帧缓存等。从 OpenGL 提供的所有功能来看，实时渲染功能最为突出，因此，与其说 OpenGL 是一种建模工具，还不如说它是一种渲染工具。

纵观 OpenGL 提供的所有功能及工具，我们找不到关于与外部建模工具交互的相关函数。事实上，OpenGL 强有力的图形函数本身并不要求开发者将模型数据写成固定格式，因此，OpenGL 不提供对其他建模工具或 CAD 软件生成的数据文件的支持。这样一来，对于复杂的图形应用程序，尤其是可视化仿真及演示系统等应用，程序员就不得不花费大量的时间，逐条语句地完成建模工作。

利用 AutoCAD 生成复杂物体的模型是非常简单的，这些文件可以按照 DXF 格式进行保存。如果能在 OpenGL 应用程序中直接调用这些数据文件，不仅能极大地降低开发周期及成本，更重要的是还可以提高应用程序的可扩充性能。

本章全面地分析了 DXF 文件的数据格式，并考虑了在 AutoCAD 的不同版本中 DXF 文件的细微差别，详细描述了从 OpenGL 应用程序调用 DXF 文件的程序实现细节，并给出了易于扩充的调用程序的源代码。在本章的最后，给出了一个应用实例。主要章节如下：

- DXF 格式文件分析。
- DXF 接口程序的编写。
- OpenGL 的 DXF 扩展库函数的制作。
- 应用实例。

15.1　AutoCAD 简介

AutoCAD 与 3DS MAX 是目前应用最为广泛的图形应用开发平台。

目前，利用 AutoCAD 及 3DS MAX 等建模软件进行的开发是非常广泛的，并且，这些软件对某些格式的数据文件有较一致的支持，如 DXF 文件或 3DS 文件，甚至 DXF 文件几乎成为事实上的图形文件的标准格式，广泛应用于从工作站到微机各种图形环境。

15.2　DXF 文件分析

15.2.1　DXF 文件的基本结构

DXF（Drawing eXchange Format）即图形交换文件，是一种顺序文件，它是在一定的组代码符号规定下，包括了实体命令和几何数据信息在内的数据文件。一个 DXF 文件包括了对应图形的图形数据库中的所有信息。DXF 文件有两种形式，一种是 ASCII 码形式，一种是二进制形式。

一个完整的 DXF 文件由 4 个段（SECTION）和一个文件尾组成。它们的顺序是：标题段（HEADER），表段（TABLES），块段（BLOCKS），实体段（ENTITIES）和结尾（EOF）。其总体结构如下：

```
    0
SECTION
    2                        标题段开始
HEADER
......
    0
ENDSEC                       标题段结束
    0
SECTION
    2
TABLES                       表段开始
......
    0
ENDSEC                       表段结束
    0
SECTION
    2                        块段开始
BLOCKS
......
    0
ENDSEC                       块段结束
    0
SECTION
```

```
    2                      实体段开始
ENTITIES
......
    0
ENDSEC                     实体段结束
    0
  EOF
```

在 DXF 文件中，每两行构成一组。每组中的第一行为组代码，第二行为组值。组代码是一个不超过 3 位数的正整数，组值的类型由组代码决定。下面是常用的组代码的含义：

0　　标识一个事物的开始，如一个段、一个表、一个块、一个图层或一个实体等。
1　　字符型数据的值，如文本字符串、文件名或属性值等。
2　　一个事物的名字，如段、表、块、线型或视图等。
3~5　　字符型数据的值。如文件名，关于线型的说明和属性提示等。
6　　线型名（固定的）。
7　　字型名（固定的）。
8　　图层名（固定的）。
9　　标题变量名（固定的）。
10~18　X 坐标值。
20~28　Y 坐标值。
30~37　Z 坐标值。
38　　基面高。
39　　厚度。
40~48　高度、宽度、半径、距离和比例因子等。
49　　重复的值，如线型表中定义线型的各短划线长度。
50~58　角度值。
62　　颜色值（固定的）。
66　　实体跟随标记，用于折线（PLINE）或带有属性的插入体（INSERT）。
70　　标题变量的状态、表的数量和标记等。
71　　定义字型时的对称方式和统计数量。
72　　文本或线型的对齐方式和统计数量。
73　　统计数量。
210　　Z、Y 和 Z 空间。
999　　解释行。

实体的坐标与相应的组代码 10~18，20~28，30~37 的用法是：根据实体所用到的点的数量，按组代码个位的 0、1、2...的顺序使用，如 3DLINE 需要两个点，所用的组代码是 10、

20、30 和 11、21、31。3DFACE 需要 4 个点，所用的组代码是 10、20、30，11、21、31，12、22、32 和 13、23、33。

15.2.2 实体段分析

1. 实体段结构

对于 OpenGL 调用来说，它只关心图形文件中的图形几何信息以及属性数据。在 DXF 文件中，这些信息全包含在 DXF 文件的实体段（ENTITIES）中。因此，我们在设计调用程序的时候，也只需读取 ENTITIES 段中数据即可。下面，让我们详细分析一下 ENTITIES 段中的具体数据结构。

实体段记录了每个实体的名称，所在图层的名字，线型名，颜色，基面高度和厚度，以及有关的几何数据。实体段的格式如下：

```
  0
SECTION
  2
ENTITIES
  0
XXXX                    （实体名，如 LINE）
  8                     （图层名）
XX                      （例如，AB）
  6                     （线型名）
XXXX                    （如 DASHED 等）
  62                    （颜色号）
  X                     （如，3）
  38                    （基面高度）
XX.X
  39                    （厚度）
XX.X
……                      （该实体的几何数据）
  0
XXXX                    （又一实体开始）
……
  0
ENDSEC                  （实体段结束）
```

2. 实体类型及其数据结构

实体（Entity）是 AutoCAD 预先定义的图元。表 15-1 列出了在 AutoCAD 中定义的各种

实体类型。

表 15-1　AutoCAD 中定义的实体类型

实体类型名	意义
POINT	点
LINE	直线
CIRCLE	圆
ARC	圆弧
TRACE	轨迹
SOLID	实体
TEXT	文本
SHAPE	型
3DLINE	三维直线
3DFACE	三维表面
DIMENSION	尺寸标注
PLINE	多义线
VERTEX	多义线顶点
SEQEND	子实体结束
INSERT	插入的块
ATTDEF	属性定义
ATTRIB	块属性

下面，我们列出了一些常用的实体在 DXF 文件的实体段（ENTITIES）中的数据结构，后面的文字说明为该实体的空间几何数据描述。

1）POINT（点）

```
    0                    开始
POINT
    8
AB
    10                   X 坐标
6.0
    20                   Y 坐标
4.0
    30                   Z 坐标
5.0
```

2）CIRCLE（圆）

0	开始
CIRCLE	
8	
A2	
10	圆心 X 坐标
2.5	
20	圆心 Y 坐标
5.0	
40	半径
2.0	

3）ARC（圆弧）

0	开始
ARC	
8	
A3	
10	X 坐标
8.0	
20	Y 坐标
7.0	
40	圆弧半径
3.0	
50	圆弧起始角度
30．0	
51	圆弧中止角度

关于其他的一些实体类型及数据结构，请参考相应的书籍。

至此，我们已经讨论了 DXF 文件的基本组成及实体段的数据结构。在此基础上，我们就可以着手编制程序，从 OpenGL 中调用 DXF 文件了。

15.3 DXF 接口程序的编写

在这一节中，我们将扼要介绍接口程序的基本结构和主要函数模块。

接口程序主要包括 getLine()，getColor()，findEntity()，readEntity()，drawEntity()及 oecGlLoadFile_dxf()这 6 个函数，其中，oecGlLoadFile_dxf()就是 OpenGL 与 DXF 文件的接口函数，其他 5 个函数分别用于完成接口函数的部分功能。下面对各个函数模块进行说明。

1）getLine()。函数定义如下：

```
static int getLine(FILE *fp)
{
    /* 从 DXF 文件中读入一行*/
    fgets(linbuf, BUFFER_SIZE, fp);
    if (feof(fp)) return 1;

    /* 读出组代码 */
    sscanf(linbuf, "%3d", &groupcode);

    /* 从 DXF 文件中读入一行 */
    fgets(linbuf, BUFFER_SIZE, fp);
    if (feof(fp))  return 1;

    return 0;
}
```

函数 getLine()按组从 DXF 文件读取数据，即每调用一次该函数就从 DXF 文件中读出两行数据，第一行为组代码，第二行为组值。

2）getColor()。该函数将 AutoCAD 中颜色表定义的颜色从索引转换为 OpenGL 的 RGB 值。其原型参见中国水利水电出版社网站中的源代码。

3）findEntity()。函数定义如下：

```
static int findEntity(FILE *fp)
{
    while (1)
    {
     /* 是否到文件尾 */
        if (feof(fp))   return 0;

     /* 获取组代码及组值 */
     if (getLine(fp)) return 0;

     /* 组代码标记 */
     if (groupcode == 0)     {
          if (strstr(linbuf, "EOF"))  return 0;
          if (strstr(linbuf, "SECTION")) {
```

```
            if (getLine(fp)) return 0;
        if (groupcode != 2)  continue;
        if (strstr(linbuf, "ENTITIES"))  break;
       }
     }
    }
    return 1;
}
```

该函数从 DXF 文件中找出实体（ENTITIES）段。如果找到，函数返回时，DXF 文件的文件指针位于组值“ENTITIES”之后。

4）readEntity()。函数定义如下：

```
static int readEntity(FILE *fp)
{
    while (1)
    {
     /* 是否到文件尾 */
     if (feof(fp))   break;

     /* 获取组代码及组值 */
     if (getLine(fp))  break;

     /* 组代码 */
     if (groupcode < 10) {
         switch(groupcode)
           {
         /* 标志事物的开始 */
         case 0:
          /* 将实体所对应的几何数据传入实体绘制模块 */
          drawEntity();

          if (strstr(linbuf, "EOF"))  return 0;
          if (strstr(linbuf, "ENDSEC"))  continue;

          /* 将颜色重置为缺省值 */
          curobj[0] = '\0'; curcolor = 7;
```

```
    getColor(curcolor, &curr, &curg, &curb);

    /* 以下为组代码 1~9 所对应的内容，
     *  开发者可视需要进行改写 */
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
    case 6:
    case 7:
    case 8:
    case 9:
        break;
    }
}
 /* 接下来是实体对应的几何数据信息，
  * 针对不同的组代码，进行不同的处理 */

/*  X 坐标 */
else if (groupcode >= 10 && groupcode < 19)
    sscanf(linbuf, "%f", &(xcoords[groupcode-10]));
/*  Y 坐标 */
else if (groupcode >= 20 && groupcode < 29)
    sscanf(linbuf, "%f", &(ycoords[groupcode-20]));
/*  Z 坐标 */
else if (groupcode >= 30 && groupcode < 38)
    sscanf(linbuf, "%f", &(zcoords[groupcode-30]));
/* 基面高度 */
else if (groupcode == 38)
{
    //此处由开发者进行相应的处理
}
/* 实体厚度 */
else if (groupcode == 39)
```

```
{
    //此处由开发者进行相应的处理
}
/* 高度、宽度、半径、距离和比例因子 */
else if (groupcode >= 40 && groupcode < 49)
    sscanf(linbuf, "%f", &(floats[groupcode-40]));
/* 重复值 */
else if (groupcode == 49)
{
     //此处由开发者进行相应的处理
}
/* 角度值 */
else if (groupcode >= 50 && groupcode < 59)
     sscanf(linbuf, "%f", &(angles[groupcode-50]));
/* 颜色代码 */
else if (groupcode == 62)
{
    sscanf(linbuf, "%6d", &curcolor);
    getColor(curcolor, &curr, &curg, &curb);
}
/* "实体跟随" 标志 */
else if (groupcode == 66)
{
     //此处由开发者进行相应的处理
}
/* 整数值，如重复计位、标志为、或模式 */
else if (groupcode >= 70 && groupcode < 79)
    sscanf(linbuf, "%d", &(ints[groupcode-70]));
/* X, Y, Z 空间 */
else if (groupcode == 210 || groupcode == 220 || groupcode == 230)
     {
        //此处由开发者进行相应的处理
     }
 }
 return 0;
```

```
}
```

函数 readEntity()是接口程序的主要模块。通常情况是对组代码为 0 的实体进行绘制，但其他数据对实体的属性有重要的影响。在本模块中，由组代码大于 10 的各部分来读取实体的几何数据，并将其记录下来，在 drawEntity()中进行绘制。

5）drawEntity()。函数的框架如下：

```
static void  drawEntity(char *curobj)
{
 // 此处为变量定义


    /* 二个 back-to-back 三角形*/
    if (strstr(curobj, "TRACE"))
{
    // 此处完成实体“TRACE”的相关绘制操作
    glBegin();
    // 具体绘制代码
    glEnd();
    }
    else if (strstr(curobj, "SOLID"))
{
    // 此处完成实体“SOLID”的相关绘制操作
    // 与上面的方法类似
    }
    else if (strstr(curobj, "3DFACE"))
{
    //根据实体“3DFACE”的相关数据完成绘制
    // 与上面的方法类似
    }
    else if (strstr(curobj, "POLYLINE"))
    {
        //根据实体“POLYINE”的相关数据完成绘制
    // 与上面的方法类似
}
//下面对其他实体："LINE"，"POINT"，"CIRCLE"，"ARC"，
// "TEXT"，"SHAPE"，"BLOCK"，"ENDBLK"，"INSERT"，"ATTDEF"，
// "ATTRIB"，"3DLINE"，"DIMENSION"，完成相应的绘制操作。
```

```
else {
  //相应的绘制操作
  // 与上面的方法类似
     }
}
```

函数 drawEntity()是绘制实体的基本框架。在函数中列出了 AutoCAD 中的各种实体，具体的实现代码，因为篇幅关系未在本章列出。需要的读者，可从中国水利水电出版社网站中查阅相应的代码。

6）oecGlLoadFile_dxf()。本函数实现从 OpenGL 中调用 DXF 格式的数据文件，其定义如下：

```
extern int  oecGlLoadFile_dxf (char *fileName)
{
    FILE     *dxfFile    = NULL;

    /*  打开图形数据文件 */
    if ((dxfFile = fopen(fileName,"r")) == NULL)
          return 0;
    /*  调用功能模块完成实体的绘制 */
    while (findEntity(dxfFile))
        readEntity(dxfFile);

    /* 关闭 DXF 数据文件 */
    fclose(dxfFile);

    return 1;
}
```

该函数的结构相当清晰，先打开数据文件，然后调用功能模块完成实体的绘制，最后关闭文件，退出。

至此，接口程序的基本模块及功能基本完成了。读者可将网站上的程序源代码复制到自己的目录下，编译成目标文件，生成 lib 文件，以供应用程序随时调用。

15.4 OpenGL 的 DXF 扩展库函数的制作

假定包含上述代码的文件名为 oglLoadDXF.c，我们利用 VC++ 6.0 将其编译为目标代码，并制作 lib 文件，以供应用程序随时调用。下面是制作 lib 文件的基本步骤。

1）进入 File 菜单。

2）选 New 项，弹出 New 对话框。

3）在 Projects 标签中，选择 Win32 Static Library 项，在 Project Name 域中键入工程文件名，如“oglLoadDXF”，在 Location 域中键入工程文件位置，选中 Create new workspace 项，在 Platforms 域中选中 Win32 项。

4）单击 OK，则新的工程文件创建完成。

5）向工程中加入源文件。在 Workspace 窗口中，激活 FileView 标签，这时窗口中出现“oglLoadDXF files”字样，将光标置于该字样上面，单击鼠标右键，弹出菜单，选中 Add files to Project ...；或者从 Project 菜单中选中 Add to Project。弹出 Insert Files to Project 对话框。在对话框的文件列表中选择需要的 C 源文件 oglLoadDXF.c，单击 OK。这样，在 FileView 窗口的“oglLoadDXF files”文件树下出现添加的文件。

6）在 Build 菜单下，选中 Build oglLoadDXF.lib 项，完成 lib 文件的制作。

在生成 lib 文件后，可将该 lib 文件复制到 VC++的 Lib 目录下，同时需要在 Include 目录下建立一个相应的头文件对函数原型进行说明，该头文件名可以是任意的，例如，可以为 oglLoadDXF.h。

15.5　高级应用实例

本节给出一个鲨鱼的动画实例，演示 OpenGL 的强大功能。

例 15-1　海底生物仿真动画程序

```
/*头文件 atlantis.h*/
#define RAD 57.295
#define RRAD 0.01745

#define NUM_SHARKS  10
#define SHARKSIZE 6000
#define SHARKSPEED 100.0

#define WHALESPEED 250.0

typedef struct _fishRec {
   float x, y, z, phi, theta, psi, v;
   float xt, yt, zt;
   float htail, vtail;
   float dtheta;
```

```
    int spurt, attack;
} fishRec;

extern fishRec sharks[NUM_SHARKS];
extern fishRec momWhale;
extern fishRec babyWhale;
extern fishRec dolph;

extern void FishTransform(fishRec *);
extern void WhalePilot(fishRec *);
extern void SharkPilot(fishRec *);
extern void SharkMiss(int);
extern void DrawWhale(fishRec *);
extern void DrawShark(fishRec *);
extern void DrawDolphin(fishRec *);

/*主程序文件 atlantis.h*/
#include <windows.h>
#include "glos.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <gl/glaux.h>

#include "atlantis.h"

fishRec sharks[NUM_SHARKS];
fishRec momWhale;
fishRec babyWhale;
fishRec dolph;

void InitFishs(void)
{
    int i;
```

```
    for (i = 0; i < NUM_SHARKS; i++) {
        sharks[i].x = 70000.0 + rand() % 6000;
        sharks[i].y = rand() % 6000;
        sharks[i].z = rand() % 6000;
        sharks[i].psi = rand() % 360 - 180.0;
        sharks[i].v = 1.0;
    }
}

void Init(void)
{
    static float ambient[] = {0.1, 0.1, 0.1, 1.0};
    static float diffuse[] = {1.0, 1.0, 1.0, 1.0};
    static float position[] = {0.0, 1.0, 0.0, 0.0};
    static float mat_shininess[] = {127.00001};
    static float mat_specular[] = {0.58, 0.68, 0.78, 1.0};
    static float mat_diffuse[] = {0.46, 0.536, 0.54795, 1.0};
    static float mat_ambient[] = {0.2, 0.4, 0.6, 1.0};
    static float lmodel_ambient[] = {0.54, 0.74, 0.74, 1.0};
    static float lmodel_localviewer[] = {0.0};
    GLfloat map1[4] = {0.0, 0.0, 0.0, 0.0};
    GLfloat map2[4] = {0.0, 0.0, 0.0, 0.0};

    glFrontFace(GL_CW);

    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);

    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
    glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, lmodel_localviewer);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
```

```
    glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_ambient);

    InitFishs();

    glClearColor(0.20, 0.325, 0.5369, 0.0);
}

void Reshape(int width, int height)
{
    glViewport(0, 0, width, height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(400.0, 2.0, 1.0, 2000000.0);
    glMatrixMode(GL_MODELVIEW);
}

static void  CALLBACK Animate(void)
{
    int i;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    for (i = 0; i < NUM_SHARKS; i++) {
        glPushMatrix();
        FishTransform(&sharks[i]);
        DrawShark(&sharks[i]);
        glPopMatrix();
    }

     /*  Animate  */
    for (i = 0; i < NUM_SHARKS; i++) {
        SharkPilot(&sharks[i]);
```

```
        SharkMiss(i);
    }
    auxSwapBuffers();
}

void Key(unsigned char key, int x, int y)
{
    switch (key) {
    case 27:
        exit(1);
    }
}

void main(int argc, char **argv)
{
    auxInitPosition(0, 0, 300, 300);
    auxInitDisplayMode(AUX_RGB|AUX_DOUBLE|AUX_DEPTH);
    auxInitWindow("编程指南例 15-1");

    Init();
    auxReshapeFunc((AUXEXPOSEPROC)Reshape);
    auxIdleFunc(Animate);
    auxMainLoop(Animate);
}

/*运动计算文件 swim.c*/
#include <math.h>
#include <gl\glaux.h>
#include "atlantis.h"

void
FishTransform(fishRec* fish)
{

    glTranslatef(fish->y, fish->z, -fish->x);
    glRotatef(-fish->psi, 0.0, 1.0, 0.0);
```

```
    glRotatef(fish->theta, 1.0, 0.0, 0.0);
    glRotatef(-fish->phi, 0.0, 0.0, 1.0);
}

void
SharkPilot(fishRec *fish)
{
    static int sign = 1;
    float X, Y, Z, tpsi, ttheta, thetal;

    fish->xt = 60000.0;
    fish->yt = 0.0;
    fish->zt = 0.0;

    X = fish->xt - fish->x;
    Y = fish->yt - fish->y;
    Z = fish->zt - fish->z;

    thetal = fish->theta;

    ttheta = RAD * atan(Z / (sqrt(X * X + Y * Y)));

    if (ttheta > fish->theta + 0.25) {
        fish->theta += 0.5;
    } else if (ttheta < fish->theta - 0.25) {
        fish->theta -= 0.5;
    }
    if (fish->theta > 90.0) {
        fish->theta = 90.0;
    }
    if (fish->theta < -90.0) {
        fish->theta = -90.0;
    }
    fish->dtheta = fish->theta - thetal;

    tpsi = RAD * atan2(Y, X);
```

```
fish->attack = 0;

if (fabs(tpsi - fish->psi) < 10.0) {
    fish->attack = 1;
} else if (fabs(tpsi - fish->psi) < 45.0) {
    if (fish->psi > tpsi) {
        fish->psi -= 0.5;
        if (fish->psi < -180.0) {
            fish->psi += 360.0;
        }
    } else if (fish->psi < tpsi) {
        fish->psi += 0.5;
        if (fish->psi > 180.0) {
            fish->psi -= 360.0;
        }
    }
} else {
    if (rand() % 100 > 98) {
        sign = 1 - sign;
    }
    fish->psi += sign;
    if (fish->psi > 180.0) {
        fish->psi -= 360.0;
    }
    if (fish->psi < -180.0) {
        fish->psi += 360.0;
    }
}

if (fish->attack) {
    if (fish->v < 1.1) {
        fish->spurt = 1;
    }
    if (fish->spurt) {
        fish->v += 0.2;
```

```
        }
        if (fish->v > 5.0) {
            fish->spurt = 0;
        }
        if ((fish->v > 1.0) && (!fish->spurt)) {
            fish->v -= 0.2;
        }
    } else {
        if (!(rand() % 400) && (!fish->spurt)) {
            fish->spurt = 1;
        }
        if (fish->spurt) {
            fish->v += 0.05;
        }
        if (fish->v > 3.0) {
            fish->spurt = 0;
        }
        if ((fish->v > 1.0) && (!fish->spurt)) {
            fish->v -= 0.05;
        }
    }

    fish->x += SHARKSPEED * fish->v * cos(fish->psi / RAD) * cos(fish->theta
/ RAD);
    fish->y += SHARKSPEED * fish->v * sin(fish->psi / RAD) * cos(fish->theta
/ RAD);
    fish->z += SHARKSPEED * fish->v * sin(fish->theta / RAD);
}

void
SharkMiss(int i)
{
    int j;
    float avoid, thetal;
    float X, Y, Z, R;
```

```
        for (j=0; j<NUM_SHARKS; j++) {
            if (j != i) {
                X = sharks[j].x - sharks[i].x;
                Y = sharks[j].y - sharks[i].y;
                Z = sharks[j].z - sharks[i].z;

                R = sqrt(X * X + Y * Y + Z * Z);

                avoid = 1.0;
                thetal = sharks[i].theta;

                if (R < SHARKSIZE) {
                    if (Z > 0.0) {
                        sharks[i].theta -= avoid;
                    } else {
                        sharks[i].theta += avoid;
                    }
                }
                sharks[i].dtheta += (sharks[i].theta - thetal);
            }
        }
    }
```

图 15-1 给出了程序的运行结果。

图 15-1　例 15-1 的运行结果

第 16 章　高级应用（二）：三维随机分形地形的生成

本章主题

在上一章，我们介绍了 OpenGL 读取一些建模工具，如 AutoCAD，所生成的三维图形文件的具体实现过程。虽然，先建模后调用这一基本做法为我们的实际图形应用的实现提供了一种选择，但正如我们要在本章中介绍的一样，我们还有其他的选择。

实际上，如果在一个三维视景仿真系统中，要将所有需要的地形作为文件存储，其代价非常高昂，至少在存储空间的占用方面确实是一笔不小的开销。在本章中，我们利用数学上的分形原理来生成复杂的三维地形。虽然学习分形的数学原理可能会耗费像本人这样的程序员的半生精力，但在这里我们需要了解的只是一些皮毛。在这里，学会利用分形原理生成复杂的地形也许只会花费大家半小时的时间。

在本章中，我们要先花一点时间了解一下分形的基本概念，比如什么是自相似等等。在此基础上，我们给出了一个利用分形生成三维地形的完整程序，并对必要的地方进行了较为详细的解释。

16.1　分形的基本原理

1986 年，在计算机图形专业会议 SIGGRAPH 上，Gavin S. P. Miller 在其论文 Definition and Rendering of Terrain Maps 中描述了如何利用分形生成复杂地形的算法，利用这些算法生成的风景图几乎达到了以假乱真的程度，并由此掀起了一场计算机图形的革命。

分形的数学原理相当复杂，甚至令人费解。然而，完全理解这些数学原理并不是掌握这些分形算法的必要条件。否则在我们解释算法之前将涉及到的分形数学原理可能就会耗费掉多数人半生的时间。鉴于这样的原因，在本章中我们不打算深入到数学细节中，也不对分形的应用领域进行探讨。我们的重点是描述分形地形生成背后的基本概念，介绍一下较为著名的“Diamond-Square”算法。然后利用该算法演示如何静态拼嵌高度数据数组，因为这些数据可用于几何地形数据、地形纹理数据及云纹理映射等很多方面。

分形究竟有什么用呢？显而易见，在这里我们要讨论分形在地形模拟方面的应用，而随机地形图对飞行模拟或制作背景纹理图十分有用。生成地形的算法同样也可用于生成部分云天的纹理图。

需要指明的是，我们在本章中讨论的算法纯粹是侧重于如何生成地形，如果为游戏应用而需要能更加快速地绘制地形的算法，则可能需要参考相关资料。并且，我们只描述生成地

形模型的过程，如何渲染则需要读者自己努力。

16.1.1　自相似

任何分形最关键的概念是自相似。当一个物体的一部分放大后看起来仍与整个物体一样，那这个物体就是自相似。

考虑一下人体的循环系统。这是自然界中自相似的一个典型例子。从最大的动脉和静脉分支直到最小的毛细血管，整个过程都显示了相同的分支模式。在显微镜下所看到的毛细血管可能和大动脉没有两样。

现在再考虑一个简单的球体。可能大多数朋友会认为它是自相似的。其实不然！如果我们将球体大幅度放大，它看起来就不再像一个球了，而更像一块平板。比如，我们能够感觉到的地球，如果不在太空中，你绝对看不出是一个球体。球体不是自相似的，它最适宜用传统的欧几里德几何而不是分形几何进行描述。

地形属于自相似范畴。放在手掌上的碎岩石锯齿状边缘与远处地平线边的山脊有相同的不规则形状。这就使得我们可以用分形来生成地形，不管显示时怎么放大，它看起来仍然像地面。

16.1.2　一维中点变换

后边要讲的 Diamond-Square 算法，是在二维平面上使用中点变换的一种算法。为了帮助理解，我们先看一维的情况。

当山脉出现在远处地平线处时，利用一维中点变换是绘制山脊的最好算法。我们看看该算法的基本原理。

```
以一条水平地平线段开始
重复足够多次{
    对场景中的每条线段做{
        找到线段的中点
        在 Y 方向上随机移动中点一段距离
        减小随机数取值范围
    }
}
```

将随机数值域减小到什么范围比较合适呢？那取决于分形地形的陡峭程度。每次循环减少得越多，所得山脊线就越平滑。但如果减得太多，则会有明显的锯齿感。实际的应用中可以将粗糙度存放在一个常量里。在后面我们会解释如何能够掌握粗糙度。

来看个例子。我们以一条 x 从-1.0 到 1.0，y 均为 0 的线段开始。我们将随机值范围设为-1.0 到 1.0（可任意取）。我们在此范围里生成一个数字，并将中点移动这么多。这一步操作完成后，我们就得到了如图 16-1 所示的形状。

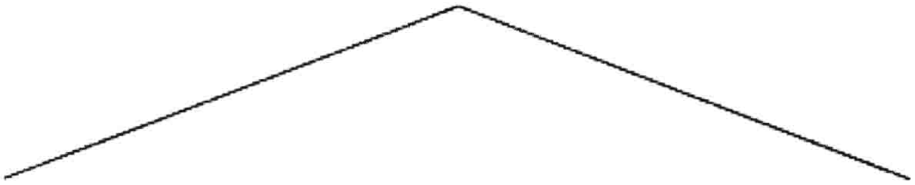


图 16-1　一维变换（第一次）

现在第二次经过外圈循环，我们有两条线段，长度均为原来的一半。我们的随机值也减半，即-0.5 到 0.5。我们为两个中点都生成一个这个范围内的随机点，结果如图 16-2 所示。

图 16-2　一维变换（第二次）

再次缩减范围，现在是 -0.25 到 0.25 。再以该范围内的数变换四个中点后，我们得到了如图 16-3 所示的结果。

图 16-3　一维变换（第三次）

在这个过程中，有两点是值得注意的。

首先，该算法是递归的。实际上，它可以用一个迭代过程来实现。对于上述情况，递归或迭代都可以。对于二维表面的生成，使用迭代实现比递归更好一些。为了保持一致性，我们在一维线段和二维表面的相应代码中都使用迭代算法。

其次，上面的算法是一个非常简单的算法，而我们可以利用该算法创建非常复杂的结果。这正是分形算法的美妙之处，使用一些简单的指令可以建立一个具有丰富细节的图像。

目前，少量简单的指令集能够生成复杂图像的事实已经成为一个新的研究领域，称之为分形图像压缩。其思想是保存建立图像的递归指令而不是保存图像本身。这对于自然界的分形图像是极有用的，因为指令相对图像占用的空间要少得多。

我们将上面例子中的数据输出到一个绘图程序，可以得到类似图 16-4 的画面。

图 16-4　一维变换生成的分形地形

在介绍二维分形表面之前，首先需要了解粗糙度常量的概念。这个常量决定了每次循环中随机数值域的减少量，也就是说，它决定了分形结果的粗糙程度。在后面的例子中，粗糙度取的是 0.0 到 1.0 之间的浮点数，并称之为 H。因此 pow(2.0, -h)是 1.0（对于小 H）到 0.5（对于大 H）范围内的数。随机数的范围在每次循环时乘上这个值。如果 H 设为 1.0，则随机数范围将每次循环减半，从而得到一个非常平滑的分形。将 H 设为 0.0 ，则范围根本不减小，结果有明显的锯齿感。

下边是三个山脊，分别用不同的 H 值绘制，如图 16-5 所示。

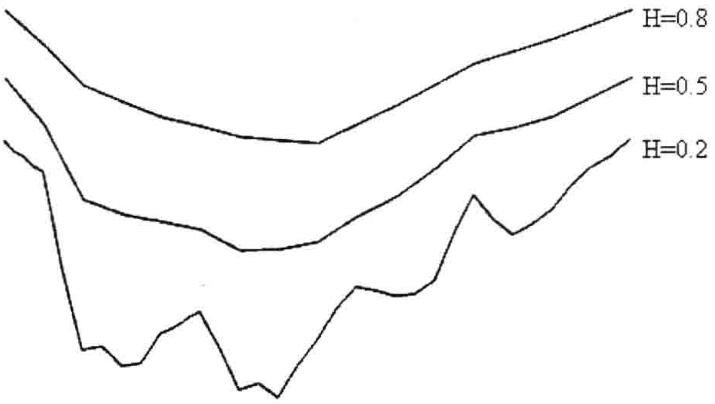


图 16-5　粗糙度的概念

上面介绍的中点变换算法可以用一个一维数组实现，数组成员是表明线段端点垂直位置的高度值。这个数组就是一个一维高度图。它将索引（x 值）映射为高度值（y 值）。

为了模拟随机地形，我们将该算法推广到三维空间。要实现这一点，我们需要一个二维高度值数组，它将索引（x,z）映射为高度（y）。数组只需保存高度值（y）。水平面值（x 和 z）可以在分析数组时及时生成。

通过对每个高度指定一个颜色，可以将一幅高度图显示为一幅图像。例如图 16-6 所示图像，高处为白色，低处为黑色。

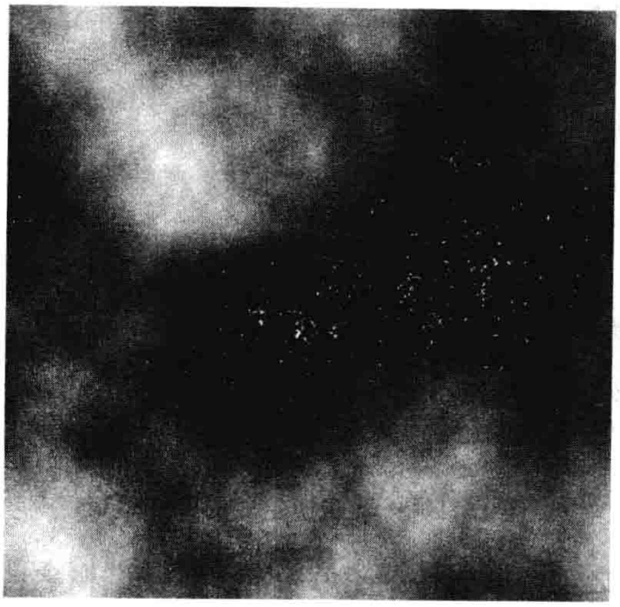


图 16-6 分形纹理图

绘制高度图的方法对于生成云彩纹理图是很有用的，后边还会深入讨论。现在我们介绍如何拼嵌我们的二维高度图数组。

16.1.3 Diamond-Square 算法

虽然 Miller 在其论文中称 Diamond-Square 算法存在着固有的缺陷，但我们还是先介绍他在论文中所采用的随机地形的生成方法。

Miller 在论文中用 Diamond-Square 算法建立了一座山，地形图上的山峰是通过人为增加网格中心点的高度来产生的。他让数组中所有的点都随机生成。如果 Miller 只简单地随机生成中心点，那么即使是他也会同意该算法是个经典的地形生成器。Diamond-Square 算法最先是由 Fourniew，Fussell 和 Carpenter 提出的。

我们先考虑从一个很大的空二维数组开始。这个数组应该取多大呢？为简化起见，假定该数组是方的，维数应该是 2 的 n 次方加 1（如 33×33，65×65，129×129 等）。将四个角设为相同高度。这时候，我们所得到的东西是一个正方形。

举一个简单的例子。选取一个 5×5 的数组，如图 16-7 所示，图 a 的四个角得到了初始高度值，用黑点表示。

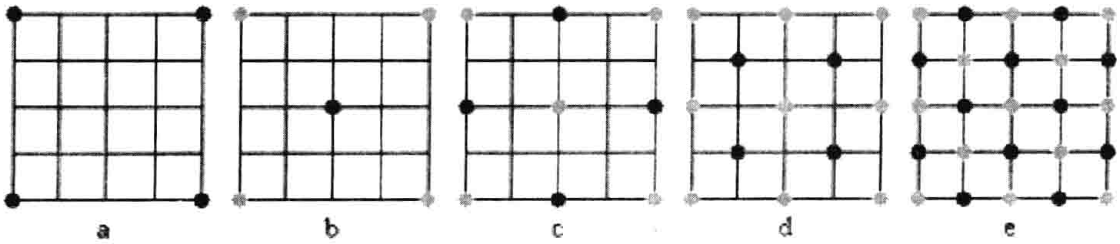


图 16-7 Diamond-Square 算法的实现过程

地形的生成是一个递归过程，该过程分两步：

1）diamond 步：取相邻四个点组成的正方形，在正方形的中点生成一个随机值（中点为两对角线交点）。中点值是平均四个顶点的值再加上一个随机量计算得到的。这样就得到了一个棱锥。当网格上分布着多个正方形时有点像钻石。

2）square 步：取相邻四点形成的棱锥，在棱锥的中心生成一个随机值。平均顶点值再加上与 diamond 步相似的随机量，计算出每条边中点值。这又给出一个正方形。

这样，如果已经生成了一个种子正方形并经过单独一次细分过程将得到四个方形。第二次经过该过程得到 16 个方形，第三次得到 64 个方形。增长得很快。方形数目等于 2^{2I}，其中 I 为递归经过细分过程的次数。

图 16-8 示意了使用两次 Diamond-Square 算法所得到的结果。

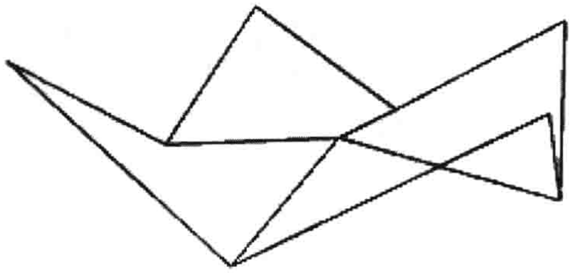


图 16-8　两次使用 Diamond-Square 算法的结果

对于第一遍经过 diamond 步时，我们依据四个角的值在数组中心生成一个值。我们平均四个角的值（如果种子值相等则完全没必要），并加上一个-1.0 到 1.0 之间的随机值。在图 16-7（b）中，新值显示成黑色，已经存在的点显示为灰色。

对于 square 步，我们在相同的范围内生成随机值。这一步时有四个棱锥；它们在数组中心相交，这样我们计算四个 diamond 中心。diamonds 的角被平均以找出新值的基数。图 16-7（c）用黑色显示新值，现存值为灰色。

以上是第一遍，如果用线将这 9 个点连起来，就可以得到一个线框的表面，看起来如图 16-8 所示。

现在进行第二遍。再次从 diamond 步开始。第二遍与第一遍有两点不同。首先，我们现在有四个四边形面不是一个，因此我们得计算四个方面的中心。其次，这是关键，生成随机数的范围已经被减小了。因为例子的缘故，让我们认为正在使用一个 H=1.0 的值。这样我们的随机数取值范围将为从（-1.0,1.0）到（-0.5,0.5）。在图 16-7（d）中，我们这一步计算得到的四个正方形中心值显示为黑色。

最后，我们进行第二遍的 square 步。有 12 个棱锥中心，我们现在需要计算 12 个新值，如图 16-7（e）中黑色点所示。

现在数组中全部 25 个元素都已经生成。我们可以得到如图 16-9 所示的线框曲面。

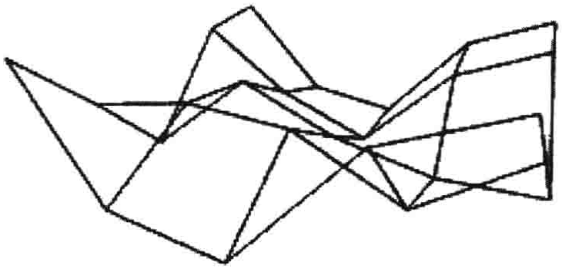


图 16-9 三次使用 Diamond-Square 算法的结果

如果分配更大的数组，我们可以进行更多遍计算，每一遍加入更多细节。例如，5 遍之后表面看起来如图 16-10 所示。

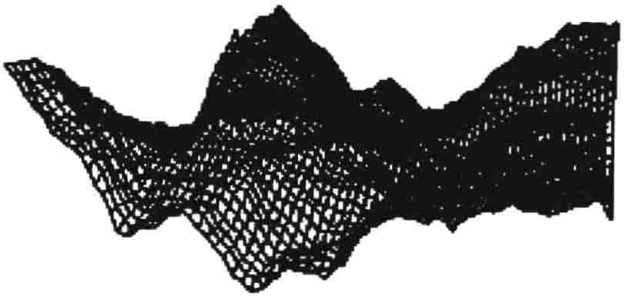


图 16-10 多次使用 Diamond-Square 算法的结果

前面提到过，数组维数需要为 2 的整数次方加 1。这是因为二维数组中的浮点数必须等于 $(2^n+1)^2$。8 次迭代将需要一个 257×257 的浮点数组，对于标准的 32 位 IEEE 浮点数来说超过 256K 内存。

现在我们知道如果采用浮点数要占用很多内存资源。如果用 char 取代 float，资源的消耗要省一些。在后面的例程中，我们使用了 float 类型。如果读者对内存很关心，那么可以使用 char 。

例程演示了处理尺寸的另外一种方法。用一个大数组依据 diamond-square 算法进行定位及拼嵌。然后从平行投影体顶视图进行绘制。该图像最后被用作第二个数组上的纹理图。纹理图如图 16-11 所示。

该图经过人工着色，山峰为白色，山谷为绿色，两者之间为灰色。读者可以利用例程源码试试自己的配色方案。

前面提到，用迭代算法实现这个例程比递归要好一些。原因是，一个递归实现可能会采用如下形式：

1）执行 diamond 步。

2）执行 square 步。

3）减小随机数范围。

4）调用自己四次。

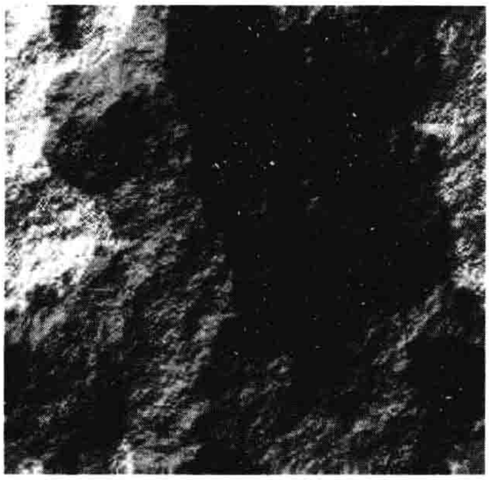


图 16-11　使用 Diamond-Square 算法生成的纹理

这种算法相当简洁，而且毫无疑问它也能工作。但它要求用不充分的数据生成某些点。实际上，我们可以看到，经过第一遍 diamond 运算之后，还将调用 square 算法，但这时并没有一个棱锥的四个顶点的全部数据。

因此，我们还是用迭代算法实现。伪代码如下：

```
当 square 边长度大于 0 时{
遍历数组，对每个正方形表达执行 diamond 步
遍历数组，对每个棱锥表达执行 diamond 步
减小随机数范围
}
```

这样就消除了递归实现中出现的棱锥顶点数据丢失问题。但在生成数组边界的点时还是会碰到这个问题。在图 16-12 中，数组中构成棱锥顶点的位置用亮灰色表示。它们应该被平均以找出新的基本值，即图中黑色的点。

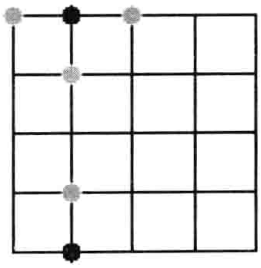


图 16-12　Diamond-Square 算法的标记

注意用黑色标记的两个值。它们实际上是相同的值。每次你在 square 步计算一个边界上的值时，记得同时把它保存在数组对面边上。

这意味着图 16-7（e）中，我们实际上不必计算 12 个单独的值，因为其中的四个在数组相对的两条边上重复。实际上只有 8 个值需要计算。

感兴趣的读者可以练习一下。取出源代码并使用它在边界的值不重复时也能工作，但这对算法正常工作是没有必要的，按照本节的方式去做就可以了。

如果读者还没运行过例程，现在可以打开看看了。它从两次迭代生成的曲面开始。曲面是用线框绘制的，只是简单地将数组中的值用线段接起来。数组中的值被当作 Y 值，而 X 和 Z 坐标是在数组分析时即时生成的。通过将一个方形分成两个三角形可以轻易地使用三角形绘制出这个曲面。三角形通常都是很好的图元，因为它总是凸性的平面。

用 View Options 对话框调节 Random seed 值。这会导致生成不同的曲面。增加 iterations 值给曲面增加细节。在代码中，我们将这个值限制到 10，实际也是考虑到目前大多数计算机的内存问题。

第一个 H 值控制表面粗糙度，默认值为 0.7。读者可以调整该值，看看会有什么结果。

不过，上述算法偶尔会产生局部尖刺或皱折，这也是今后需要改进的地方。但总的来说，该算法能够满足绝大多数应用的需要。

16.1.4 蓝天白云的生成

通过前面的叙述，现在我们知道了如何生成表面。比如，我们可以生成大量的三角形，并进行渲染；也可以生成一个高分辨率的纹理图，并将它贴到低分辨率的表面上。所有这些都能取得相当好的效果。但是，我们如何来生成头上的天空呢？通过本节的阅读，你会发现这个工作比大多数人想象的都要简单得多。

用 Diamond-Square 算法拼嵌完成的数组非常适于表示蓝天的纹理图。与把数组看作一套高度图上的 y 值相反，我们把它看成云的不透明度数据。最小数组值代表最蓝，天空中最清晰的部分，最大的值代表最白，天空中云最重的部分。

接下来我们会看到，虽然算法简单，但要分析数组并生成如图 16-13 所示的纹理图仍然是相当琐碎的。

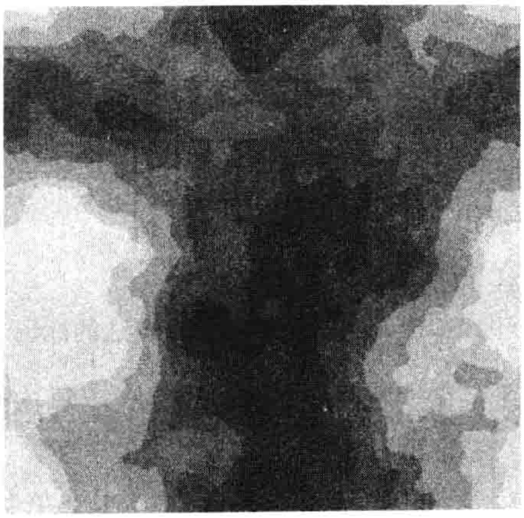


图 16-13 Diamond-Square 算法生成的蓝天白云图

这与前面的高度图很像，但我们已经限定了高、低值，并用它来建立蓝天白云的天空纹理图。后面的例程中包含了生成蓝天白云的功能。将 Select rendering type 设置为 2D mesh/clouds。使用默认值时的效果不很好，看起来蓝天白云图像离散，如果调整 Cloud iterations 的值，效果会发生变化。

读者或许会有自己的想法，也可以在源代码中增加更多的控制功能。例如，可以用选定的值来初始化种子值，这样，可以得到形态各异的三维地形图。

修改代码，使之在赋值时跳过已有值的数组成员是非常容易完成的。例如，初始化数组为-10.0，并在前几轮指定自己的值作为种子，再修改分形生成代码，只给当前值为 -10.0 的赋值。这样，前几轮将不会生成任何值，因为你的种子值已经在那儿了。后续的迭代将在种子值基础上生成新值。

如何取得种子值呢？如果想要的形状遵循某个已知的数学公式，如正弦曲线，就用这个函数生成值。否则，你得找出创造性的方法完成。

除了 Diamond-Square 算法，还有许多方法可以用于表面的拼嵌。

用连续随机增加的办法，给二维数组的一个随机部分增高一个很少的量。反复多次，对所选中的数组区域加上一个很小的随机值。这种办法可以生成相当不错的结果，但不是线性计算的。如果计算时间无所谓，那么建议试试这种算法。

另一个相似的方法是制造一个穿过数组的“折断”，并增加其中的一边，就像地震出现一样，再重复多次。这也不是一个线性算法，要多遍才能取得可以接受的结果。

16.2　分形地形生成实例

16.2.1　程序的运行

本程序的源代码使用 OpenGL API 绘制。例子中的核心代码将在本节列出。假定本程序的执行文件名为 Fractal，运行后界面如图 16-14 所示。

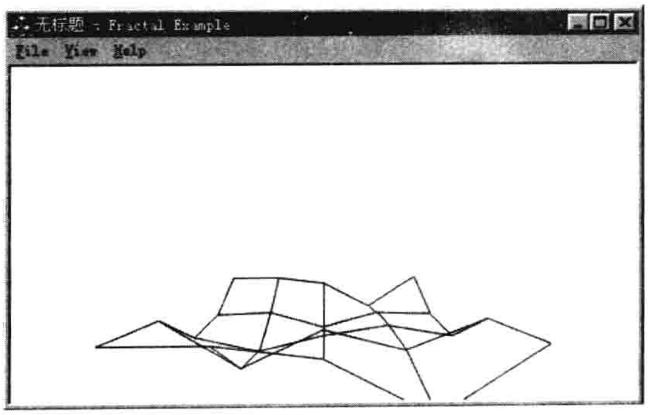


图 16-14　分形程序初始界面

打开 View Options 对话框，从 Select render type 下拉菜单选择 2D mesh/rendered，将 Iterations 值改为 4，将 Tile 值改为 3，得到如图 16-15 所示的结果。

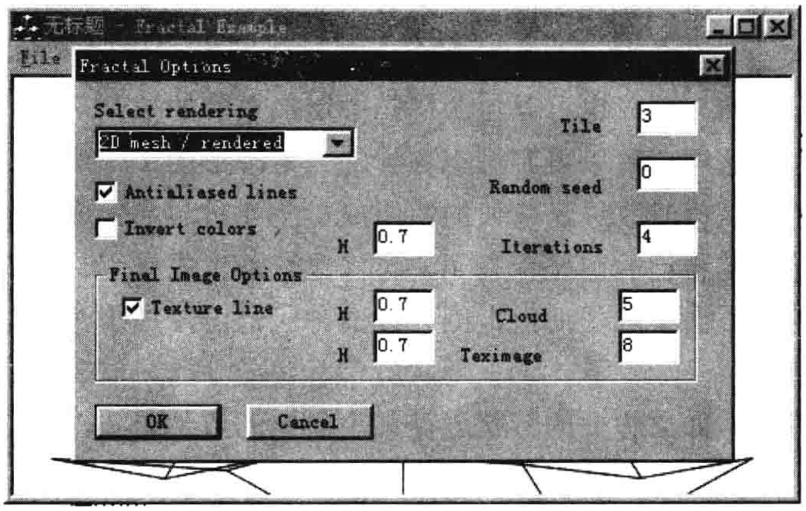


图 16-15　分形程序的选项设置

保持上述选项，按下确定按钮，得到如图 16-16 所示的结果。

图 16-16　渲染后的分形地形图

16.2.2　代码结构分析

fractmod.c 和 fractmod.h 是本例所有程序的核心 C 代码，它构成分形生成模块。

CFractalExampleView 类是从 COpenGLView 类衍生的。COpenGLView 类最初由 Ron Fosner 写成。细节可参考他所著的 OpenGL programming for Windows 95 and Windows NT 一书（Addison-Welsley 出版社出版）。

COpenGLView 有一个 RenderScene 虚成员，我们在 CFractalExampleView 里将其重载。在这里将完成主要的绘制工作。函数先检查 Rendering type 的设定。当设为 2D mesh/lines 或

1D midpoint displacement 时，工作由 RenderScene 完成。否则，该项工作由其他函数来完成。

CFractalExampleView::OnViewDialog 生成 View Options 对话框，并处理设置，同时在 CFractalExampleView 与对话框类间提取数据。

CFractalExampleView::OnInitialUpdate 负责将所有 CFractalExampleView 成员变量设置为默认值（含对话框值）。

在 FractalExampleView.cpp 文件中，有一个名为 DEF_HIGHT_VALUE 的预处理常量。它传给 fractmod.c 文件中的分形生成函数，用来缩放高度值。其实它应该是由 View Options 对话框控制的变量。

下面我们列出了该例子中的关键代码。

1. 地形排列的边界拼接算法

```
z = -1.f * (float) (tile-1);
    for (i=0; i<tile; i++) {
    x = -1.f * (float) (tile-1);
    for (j=0; j<tile; j++) {
        glPushMatrix ();
        glTranslatef (x, 0.f, z);
        draw2DFractArrayAsLines (surfFA, size);
        glPopMatrix ();
        x += 2.f;
    }
    z += 2.f;
}
```

2. 地形绘制

```
    bool RenderScene()
{
    int size = 1 << iterations;
    GLfloat black[4] = {0.f, 0.f, 0.f, 1.f};
    GLfloat white[4] = {1.f, 1.f, 1.f, 1.f};

    // Handle inverted colors.
    if (colorInvert)
        glClearColor (black[0], black[1], black[2], black[3]);
    else
        glClearColor (white[0], white[1], white[2], white[3]);
```

```
// Handle non line mode renderMode settings.
if (renderMode == twoDR_clouds)
return (renderCloudMap ());
else if (renderMode == twoDR_teximage)
return (renderTeximageMap ());
else if (renderMode == twoDR_rendered)
return (renderFullImage ());

// Shouldn't take too long, unless drawing huge
// AA scenes, but what the heck...
CWaitCursor hourglass;

glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
if (arrayIs2D)
// Needs to be identical to the same call in renderFullScene.
    gluPerspective (60., aspectRatio, .1, 15.0);
else
glOrtho (-aspectRatio, aspectRatio, -1., 1., 1., -1.);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
if (arrayIs2D) {
    glTranslatef (0.f, yTrans, zTrans);
glRotatef (yRot, 0.f, 1.f, 0.f);
}

glPushAttrib (0xffffffff);

glDisable (GL_DEPTH_TEST);
glDisable (GL_LIGHTING);
if (colorInvert)
glColor4fv (white);
else
glColor4fv (black);
```

```
if (aaLines) {
glEnable (GL_LINE_SMOOTH);
glEnable (GL_BLEND);
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
}
else {
glDisable (GL_LINE_SMOOTH);
glDisable (GL_BLEND);
}

glClear (GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);

if (arrayIs2D) {
float x, z;
UINT i, j;

// validate surface
if (surfFA && surfFAdirty) {
    freeFractArray (surfFA);
    surfFA = NULL;
}
if (!surfFA) {
    surfFA = alloc2DFractArray (size);
    if (surfFA==NULL) return (FALSE);
    fill2DFractArray (surfFA, size, randomSeed,
        DEF_HEIGHT_SCALE, surfaceH);
    surfFAdirty = FALSE;
}

z = -1.f * (float) (tile-1);
for (i=0; i<tile; i++) {
    x = -1.f * (float) (tile-1);
    for (j=0; j<tile; j++) {
        glPushMatrix ();
```

```
            glTranslatef (x, 0.f, z);

            draw2DFractArrayAsLines (surfFA, size);

            glPopMatrix ();
            x += 2.f;
        }
        z += 2.f;
    }
    }
    else {
    float x;
    UINT i;
        float    *fa;

    // Always validate and free 1D array.
    // We won't be doing any realtime transformations on it.
    fa = alloc1DFractArray (size);
    if (fa==NULL) return (FALSE);
    fill1DFractArray (fa, size, randomSeed,
        DEF_HEIGHT_SCALE, surfaceH);

    x = -1.f * (float) (tile-1);
    for (i=0; i<tile; i++) {

        glPushMatrix ();
        glTranslatef (x, 0.f, 0.f);

        draw1DFractArrayAsLines (fa, size);

        glPopMatrix ();
        x += 2.f;
    }

    freeFractArray (fa);
```

```
    }

    glFlush();
    glPopAttrib ();

    return (TRUE);
}
```

3. 纹理

```
    bool renderTeximageMap ()
{
    CWaitCursor hourglass;
    int  size = 1 << teximageIter;
    UINT pmapDim, smallDim;

    if (teximageFA && teximageFAdirty) {
    freeFractArray (teximageFA);
    teximageFA = NULL;
    }
    if (!teximageFA) {
    teximageFA = alloc2DFractArray (size);
    if (teximageFA==NULL) return (FALSE);
    fill2DFractArray (teximageFA, size, randomSeed,
        DEF_HEIGHT_SCALE, teximageH);
    }

    // Find the biggest power of two that will fit
    // in the current window. This is where we will
    // draw the image.
    smallDim = (currentWidth > currentHeight) ?
    currentHeight : currentWidth;
    pmapDim = 1 << (UINT) (log ((float) smallDim) / log (2.));

    glPushAttrib (0xffffff);

    // Only draw into biggest power of 2 square.
```

```
glViewport (0, 0, pmapDim, pmapDim);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glOrtho (-1.f, 1.f,
-1.f, 1.f, 1., -1.);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
// The array is rendered into the XZ plane. We need to
// rotate by 90 degrees so we will be looking at it
// "top down".
glRotatef (90.f, 1.f, 0.f, 0.f);
// Init lights now. Light position must be specified
// after the rotate, or else it won't be rotated.
initLights ();

glDisable (GL_DEPTH_TEST);
glDisable (GL_FOG);
glEnable (GL_LIGHTING);
glDisable (GL_BLEND);

glClear (GL_COLOR_BUFFER_BIT);

renderAsTriangles (teximageFA, size, 1, 0);
glFlush ();

glPopAttrib ();



if (teximageFAdirty) {
GLubyte *pmap;

pmap = (GLubyte *) malloc (pmapDim*pmapDim*3);
if (pmap==NULL) return (FALSE);

glReadPixels (0, 0, pmapDim, pmapDim,
```

```
        GL_RGB, GL_UNSIGNED_BYTE, pmap);

    glNewList (TEXIMAGEMAP, GL_COMPILE);
    glTexImage2D (GL_TEXTURE_2D, 0, /* lod */
        3, /* num components */
        pmapDim, pmapDim, /* width, height */
        0, /* border */
        GL_RGB, GL_UNSIGNED_BYTE, /* format, type */
        pmap);

    glEndList ();

    free (pmap);
    }

    teximageFAdirty = FALSE;
    return (TRUE);
}
```

4. 云

```
    bool renderCloudMap ()
{
    CWaitCursor hourglass;
    int size = 1 << cloudIter;

    if (cloudFA && cloudFAdirty) {
    freeFractArray (cloudFA);
    cloudFA = NULL;
    }
    if (!cloudFA) {
    cloudFA = alloc2DFractArray (size);
    if (cloudFA==NULL) return (FALSE);
    fill2DFractArray (cloudFA, size, randomSeed,
        DEF_HEIGHT_SCALE, cloudH);
    }
```

```
glPushAttrib (0xfffffff);

if (cloudFAdirty) {
GLubyte *tmap;

tmap = (GLubyte *) malloc (size*size*3);
if (tmap==NULL) return (FALSE);

surfToSky (cloudFA, size, tmap);

glNewList (CLOUDMAP, GL_COMPILE);

glTexImage2D (GL_TEXTURE_2D, 0, /* lod */
     3, /* num components */
     size, size, /* width, height */
     0, /* border */
     GL_RGB, GL_UNSIGNED_BYTE, /* format, type */
     tmap);
        .
glEndList ();

free (tmap);
}
glCallList (CLOUDMAP);

glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
if (currentWidth > currentHeight)
glOrtho (-1.f, aspectRatio*2.f-1.f,
    -1.f, 1.f, 1., -1.);
else
glOrtho (-1.f, 1.f,
    -1.f, 2.f/aspectRatio-1.f, 1., -1.);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
```

```
    glDisable (GL_DEPTH_TEST);
    glDisable (GL_FOG);
    glDisable (GL_LIGHTING);
    glDisable (GL_BLEND);
    glEnable (GL_TEXTURE_2D);

    glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
    (GLfloat) GL_REPLACE);
    glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    (GLfloat) GL_NEAREST);
    glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    (GLfloat) GL_NEAREST);

    glClear (GL_COLOR_BUFFER_BIT);

    glBegin (GL_QUADS); {
    glTexCoord2f (0.f, 0.f); glVertex2f (-1.f, -1.f);
    glTexCoord2f (1.f, 0.f); glVertex2f (1.f, -1.f);
    glTexCoord2f (1.f, 1.f); glVertex2f (1.f, 1.f);
    glTexCoord2f (0.f, 1.f); glVertex2f (-1.f, 1.f);
    } glEnd ();
    glFlush ();

    glPopAttrib ();

    cloudFAdirty = FALSE;
    return (TRUE);
}
```

5. 分形计算核心代码

代码中包括了如下算法：① Square 步的顶点值平均算法；② Diamond 步的顶点值平均算法；③ 使用 Square-Diamond 算法进行分形数组填充。

源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include "math.h"

#include "fractmod.h"

#ifdef __cplusplus
extern "C" {
#endif

#ifdef _WINDOWS
/* 为 HPUX 而写，控制 Win32 编译*/
#define random() rand()
#define srandom(x) srand(x)
#endif

/*
 随机返回一个(min, max)之间的值。
 (任意一个范围被分为 32767 个值。)
 */
static float randnum (float min, float max)
{
    int r;
    float   x;

    r = random ();
    x = (float)(r & 0x7fff) /
     (float)0x7fff;
    return (x * (max - min) + min);
}

/*
 定义 randnum 的简化界面
 */
#define fractRand(v) randnum (-v, v)

/*
```

```
 计算数组中中点 i 所在线段端点值的平均值。
 *
 * Called by fill1DFractArray.
 */
static float avgEndpoints (int i, int stride, float *fa)
{
    return ((float) (fa[i-stride] +
         fa[i+stride]) * .5f);
}


/*
计算棱锥四个顶点的平均值，(i,j)为棱锥中心点，
stride 为中心到顶点的纵横间隔， size 是纵横向点个数
由于是正方形，纵横间隔及两方向点均相等。


注意，size 为纵横向顶点数，而 subSize 为线段数，均代表数组维数。
     size=subSize+1;
*/
static float avgDiamondVals (int i, int j, int stride,
                        int size, int subSize, float *fa)
{
/*     为支持排列表面无缝拼接，当(i,j)处于数组的某条边时要特别注意。
       下面头四条 if 语句处理这种情况况。最后的 else 处理一般情况（i,j 不在边界上）。
*/
    if (i == 0)
    return ((float) (fa[(i*size) + j-stride] +      //j-strid
          fa[(i*size) + j+stride] +         //j+strid
          fa[((subSize-stride)*size) + j] +   //i-strid，不存在，用对边上的顶点
          fa[((i+stride)*size) + j]) * .25f);   //i+strid
    else if (i == size-1)
    return ((float) (fa[(i*size) + j-stride] +      //j-strid
          fa[(i*size) + j+stride] +          //j+strid
          fa[((i-stride)*size) + j] +          //i-strid
          fa[((0+stride)*size) + j]) * .25f); //i+strid，不存在，用对边上的顶点
    else if (j == 0)
```

```
    return ((float) (fa[((i-stride)*size) + j] +
          fa[((i+stride)*size) + j] +
          fa[(i*size) + j+stride] +
          fa[(i*size) + subSize-stride]) * .25f);
    else if (j == size-1)
    return ((float) (fa[((i-stride)*size) + j] +
          fa[((i+stride)*size) + j] +
          fa[(i*size) + j-stride] +
          fa[(i*size) + 0+stride]) * .25f);
    else
    return ((float) (fa[((i-stride)*size) + j] +    //i-strid
          fa[((i+stride)*size) + j] +      //i+strid
          fa[(i*size) + j-stride] +        //j-strid
          fa[(i*size) + j+stride]) * .25f);   //j+strid
}

/*
 平均中心位于(i,j)的正方形四角的数据值
 stride为正方形边长度。
 与棱锥步不同，这次不会有边界的问题。
*/
static float avgSquareVals (int i, int j, int stride, int size, float *fa)
{
    return ((float) (fa[((i-stride)*size) + j-stride] +
          fa[((i-stride)*size) + j+stride] +
          fa[((i+stride)*size) + j-stride] +
          fa[((i+stride)*size) + j+stride]) * .25f);
}

/*判断size是否为2的次方，是则返回1，不是或为0则返回0。*/
static int powerOf2 (int size)
{
    int i, bitcount = 0;
    /* 下面的代码假定sizeof(int)*8可以算出一个整型数的位数。
       适用于多数平台 */
```

```
    for (i=0; i<sizeof(int)*8; i++)
    if (size & (1<<i))
       bitcount++;
    if (bitcount == 1)
    /* 注意 2 的整次方数只会有一个位为 1 */
    return (1);
    else
    return (0);
}

/*拼嵌一数组的值为近似分形布朗运动 */
void fill1DFractArray (float *fa, int size,
            int seedValue, float heightScale, float h)
{
    int  i;
    int  stride;
    int subSize;
    float ratio, scale;

    if (!powerOf2(size) || (size==1)) { return; }

    subSize = size;
    size++;

    srandom (seedValue);


/* 设定粗糙度常量
      随机数始终在 0.0 到 1.0 范围内生成。
      随机数将乘上 scale，每次迭代后 scale 将乘上 ratio 以有效地减少随机数范围。
*/
     ratio = (float) pow (2.,-h);
     scale = heightScale * ratio;

    /* 在数组中播种端点。要实现无缝拼接，这些端点相同*/
```

```
    stride = subSize / 2;
    fa[0] =
      fa[subSize] = 0.f;

    while (stride) {
        for (i=stride; i<subSize; i+=stride) {
            fa[i] = scale * fractRand (.5f) +
                avgEndpoints (i, stride, fa);
            /* reduce random number range */
            scale *= ratio;
            i+=stride;
        }
        stride >>= 1;
    }
}


/*用 diamond-square 算法拼嵌一个浮点数网格，填充二维分形高度图*/
void fill2DFractArray (float *fa, int size,
                       int seedValue, float heightScale, float h)
{
    int i, j;
    int stride;
    int oddline;
    int subSize;
    float ratio, scale;
    //只处理维数为 2 的整数次方的数组
    if (!powerOf2(size) || (size==1)) { return; }

    /* subSize 以纵横向线段数计的数组维数
       segments 以顶点数计的数组维数
    */
    subSize = size;
    size++;
```

```
    /* initialize random number generator */
    srandom (seedValue);

     /* 设定我们的粗糙度常量
        随机数始终在 0.0 到 1.0 范围内生成。
        随机数将乘上 scale，每次迭代后 scale 将乘上 ratio 以有效地减少随机数范围。
*/
     ratio = (float) pow (2.,-h);
     scale = heightScale * ratio;

    /* 设置前四个种子值。如一个 4×4 数组，我们将初始化下边*代表的点：
           * . . . *
           . . . . .
           . . . . .
           . . . . .
           * . . . *
       按 diamond-square 算法的术语，这将给我们正方形
       数组四角赋相同的值，这使我们排列数组时能无缝地拼接起来*/

    stride = subSize / 2;
    fa[(0*size)+0] =
      fa[(subSize*size)+0] =
        fa[(subSize*size)+subSize] =
          fa[(0*size)+subSize] = 0.f;

    /*现在依据棱锥种子值递增加入细节。
     根据 stride 循环,每次循环末尾都会将 stride 减半,
     直到 stride 为 0 时结束 */
    while (stride) {
         /* 先用 square 数据生成 diamond 数据。
            第一遍经过 4×4 矩阵时，已有数据为下边的 X，
            我们要生成*处的数据：
               X . . . X
               . . . . .
               . . * . .
```

```
        . . . . .
        X . . . X
    */
     for (i=stride; i<subSize; i+=stride) {
         for (j=stride; j<subSize; j+=stride) {
             fa[(i * size) + j] =
                 scale * fractRand (.5f) +
                 avgSquareVals (i, j, stride, size, fa);
             j += stride;
         }
         i += stride;
     }

     /* 先用 diamond 数据生成 square 数据。
        第一遍经过这段代码时，已有数据为下边的 X，
        我们要生成*处的数据：
          X . * . X
          . . . . .
          * . X . *
          . . . . .
          X . * . X
       i,j 代表我们在数组中的(x,y)位置。我们想生成的第一个值位于(i=2,j=0)，
        用"oddline"及"stride"来增加 j 到所需值。
    */
     oddline = 0;
     for (i=0; i<subSize; i+=stride) {
        oddline = (oddline == 0);
         for (j=0; j<subSize; j+=stride) {
             if ((oddline) && !j) j+=stride;

             /* i and j are setup. Call avgDiamondVals with the
               current position. It will return the average of the
               surrounding diamond data points. */
             fa[(i * size) + j] =
                 scale * fractRand (.5f) +
```

```
                avgDiamondVals (i, j, stride, size, subSize, fa);

            /* To wrap edges seamlessly, copy edge values around
               to other side of array */
            if (i==0)
                fa[(subSize*size) + j] =
                    fa[(i * size) + j];
            if (j==0)
                fa[(i*size) + subSize] =
                    fa[(i * size) + j];

            j+=stride;
        }
    }

    /* reduce random number range. */
    scale *= ratio;
    stride >>= 1;
  }

}


/*
 * alloc1DFractArray - Allocate float-sized data points for a 1D strip
 * containing size line segments.
 */
float *alloc1DFractArray (int size)
{
   /* Increment size (see comment in alloc2DFractArray, below, for an
      explanation). */
   size++;

   return ((float *) malloc (sizeof(float) * size));
}
```

```
/*
 * alloc2DFractArray - Allocate float-sized data points for a sizeXsize
 * mesh.
 */
float *alloc2DFractArray (int size)
{
    /* For a sizeXsize array, we need (size+1)X(size+1) space. For
       example, a 2x2 mesh needs 3x3=9 data points:

           *   *   *

           *   *   *

           *   *   *

       To account for this, increment 'size'. */
    size++;

    return ((float *) malloc (sizeof(float) * size * size));
}


/*
 * freeFractArray - Takes a pointer to float and frees it. Can be used
 * to free data that was allocated by either alloc1DFractArray or
 * alloc2DFractArray.
 */
void freeFractArray (float *fa)
{
    free (fa);
}


extern void draw2DLine (float, float, float, float);
```

```
/* 注意，要使用 draw1DFractArrayAsLines 将数组绘制成线段，要先自己定义
   四个参数的 draw2DLine，参数为两个端点的 x,y 坐标。*/
void draw1DFractArrayAsLines (float *fa, int size)
{
    int  i;
    float   x, inc;

    inc = 2.f / size;
    x = -1.f;
   for (i=0; i<size; i++) {
    draw2DLine (x, fa[i],
             x+inc, fa[i+1]);
    x += inc;
    }
}

/* 注意，要使用 draw2DFractArrayAsLines 将数组绘制成网格，要先自己定义
   六个参数的 draw3DLine，参数为两个端点的 x,y,z 坐标。*/
void draw2DFractArrayAsLines (float *fa, int size)
{
    extern void draw3DLine (float, float, float, float, float, float);
    int  i, j;
    float   x, z, inc;
    const int   subSize = size;

    size++;
    inc = 2.f / subSize;
    z = -1.f;
    for (i=0; i<size; i++) {
     x = -1.f;
     for (j=0; j<subSize; j++) {
        draw3DLine (x, fa[(i*size)+j], z,
             x+inc, fa[(i*size+j+1)], z);
        if (i<subSize)
        draw3DLine (x, fa[(i*size)+j], z,
```

```
                x, fa[((i+1)*size+j)], z+inc);
        x += inc;
    }
    if (i<subSize)
        draw3DLine (x, fa[(i*size)+j], z,
            x, fa[((i+1)*size+j)], z+inc);
    z += inc;
    }
}

/*法向计算，参数为三个顶点矢量，其叉乘积就是平面法向*/
static void genNormal (float x1, float y1, float z1,
                float x2, float y2, float z2,
                float x3, float y3, float z3,
                float *normal)
{
    float   len;
    float   v1x, v1y, v1z;
    float   v2x, v2y, v2z;
    v1x = x2 - x1;
    v1y = y2 - y1;
    v1z = z2 - z1;
    v2x = x3 - x1;
    v2y = y3 - y1;
    v2z = z3 - z1;
    normal[0] = v1y*v2z - v1z*v2y;
    normal[1] = v1z*v2x - v1x*v2z;
    normal[2] = v1x*v2y - v1y*v2x;
    len = (float) sqrt (normal[0]*normal[0] + normal[1]*normal[1] +
            normal[2]*normal[2]);
    normal[0] /= len;
    normal[1] /= len;
    normal[2] /= len;
}
```

```
/*
  用一系列带有相应法向的三角形绘制高度图。
  这是个简化的例程以便达到目的并快速运行，并演示了如何遍历一个数组。
  要使用这个例程，你必须自己定义"draw3DTriangle"函数。
  它有 12 个参数：头 9 个是三个顶点的 x,y,z 世界坐标，
  最后 3 个参数是单位长度面法向的 x,y,z 笛卡尔值。
  x,z 坐标会沿两轴覆盖-1.0 到 1.0 的网格。相应的 Y 坐标将从分形数组中取出。
  法向量将参考“上”或+y 方向生成，顶点顺序应该使用右手法则与法向保持一致。
  （右手法则：从法向量所指方向上看，顶点按逆时针排列）
 */
void draw2DFractArrayAsTriangles (float *fa, int size)
{
    extern void draw3DTriangle (float, float, float, float, float, float,
                   float, float, float, float, float, float);
    int  i, j;
    float   x, z, inc;
    const int   subSize = size;

    size++;
    inc = 2.f / subSize;
    z = -1.f;
    for (i=0; i<subSize; i++) {
     x = -1.f;
     for (j=0; j<subSize; j++) {
         float   normal[3];

         genNormal (x, fa[(i*size)+j], z,
                 x, fa[((i+1)*size)+j], z+inc,
                 x+inc, fa[(i*size+j+1)], z,
                 normal);
         draw3DTriangle (x, fa[(i*size)+j], z,
                  x, fa[((i+1)*size)+j], z+inc,
                  x+inc, fa[(i*size+j+1)], z,
                  normal[0], normal[1], normal[2]);
```

```
        genNormal (x, fa[((i+1)*size)+j], z+inc,
              x+inc, fa[((i+1)*size)+j+1], z+inc,
              x+inc, fa[(i*size+j+1)], z,
              normal);
        draw3DTriangle (x, fa[((i+1)*size)+j], z+inc,
               x+inc, fa[((i+1)*size)+j+1], z+inc,
               x+inc, fa[(i*size+j+1)], z,
               normal[0], normal[1], normal[2]);

        x += inc;
     }
     z += inc;
   }
}

#ifdef __cplusplus
}
#endi
```

附录 1　OpenGL 中的操作顺序

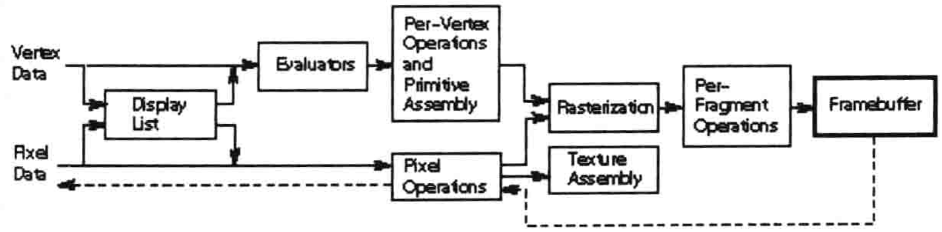
本书的第一部分描述了从开始指定顶点到片元最终被写入帧缓存之间所执行的所有操作。而且在第二部分中较详细介绍了操作次序及 OpenGL 的处理过程。本附录的介绍主要是为了便于学习。该次序并不是操作实际的执行次序。有时严格的操作执行次序实际上是没有必要的。比如说，表面可以先转换成多边形，然后再实施变换；也可以先作变换，然后再将其转换成多边形，按照这两种不同次序，结果是相同的。对于 OpenGL 的不同实现，操作顺序可能不同。本附录描述了一种可能的操作次序；不管操作次序如何，OpenGL 的任何实现都要求其结果应该是相同的。

本附录包括下列主要内容：

- 概述。
- 几何操作。
- 像素操作。
- 片元操作。
- 补充说明。

附 1.1　概述

本节给出了操作次序的概述，如附图 1-1 所示。几何数据（顶点，直线以及多边形）经历了求值程序和各顶点操作等操作过程，与此同时，像素数据（像素，图形及位图）也经历了不同的处理过程。在最终的像素数据写入帧缓存之前，几何数据和像素数据都要经过光栅化及片元操作。



附图 1-1　操作次序

所有的数据，不管是描述几何体的数据还是描述像素的数据，都要存储在显示列表中，然后进行处理。当执行显示器列表操作时，数据从显示列表中发送出去，就好像从应用程序中发送的数据一样。

所有的几何图元最终都是由顶点来描述。如果使用了求值程序，那些数据就被转化为顶点，在此之后，就被当作顶点来处理。各顶点（Per-vertex）计算是在每个顶点处进行的，接下来经过光栅化，变成片元。对于像素数据，在执行像素操作后，其结果或者存储在纹理内存中，用于多边形点划，或者被光栅化变成片元。最后，片元要经过一系列（片元）操作，将最终的像素值写入帧缓存。

附 1.2 几何操作

几何数据不管来源于显示列表、求值程序、矩形的顶点，或者来源于原始数据，都由一系列顶点及其描述的几何图元（顶点、直线或多边形）类型组成。顶点数据不仅包括（x，y，z，w）坐标，还包括法线向量、纹理坐标、颜色或颜色索引和边缘标记数据。所有这些元素，除了顶点坐标可以按任何次序指定以外，都存在缺省值。一旦调用了顶点命令glVertex*()，这些元素即被填充。如果必要的话，四维坐标（令 z=0，w=1）的所有元素的当前值就与该顶点联系起来。然后，再对全部的顶点数据进行处理。

附 1.2.1 顶点操作

在顶点操作处理阶段，每一个顶点的空间坐标通过模型视图矩阵进行变换，而法线向量则通过该矩阵的逆矩阵进行变换，如果指明的话，要重新进行归一化。如果自动纹理生成被激活，新的纹理坐标由变换后的顶点坐标生成，由它们取代原有的纹理坐标。然后，纹理坐标要通过当前纹理矩阵进行变换，然后被传递到图元装配这一步。

同时，如果光照被激活，那么光照的计算是根据变换后的顶点及法线向量坐标、当前材质、灯光以及光照模型等因素来完成的。光照计算将产生新的颜色或颜色索引，这些新的颜色或颜色索引被限制或屏蔽在合适的范围，也被传递到图元装配这一步。

附 1.2.2 图元装配

图元装配随图元，如点、线、或多边形等的不同而不同。如果单调明暗效果被激活，所有的直线或多边形的顶点的颜色及颜色索引都被设置成同一个值，如果定义并激活了特殊的裁剪平面，则它们将被用于三类的图元的裁剪（剪平面方程通过指定的模型视图矩阵的逆矩阵进行变换）。点的裁剪只是通过简单的顶点取舍处理来完成；直线或多边形的裁剪可以通过增加新的顶点来完成，这取决于直线或多边形是怎样进行裁剪的。在裁剪之后，每个顶点的空间坐标通过投影矩阵进行变换，变换的结果就以标准的视图平面 $x=\pm w$，$y=\pm w$，及 $z=\pm w$ 进行裁剪。

如果选择功能被激活，那么未裁剪掉的图元会产生一个“选择击中”报告，因此就没有必要进行过多的处理。如果没有激活选择功能，就会按 w 进行透视分割，还要进行视区和深度范围操作。还有，如果图元是多边形，则还要进行选择测试（如果激活了选择测试）。

根据多边形模式，还可将其转化为顶点或直线。

最后，点、直线、以及多边形要被光栅化为片元，这一操作过程要考虑到多边形或直线的点划、直线的宽度、以及点的尺寸等因素。光栅化涉及到在图元所占据的窗口坐标中的整数网格块的确定。片元的颜色及深度值要分配到每一个这样的方块中。

附 1.3　像素操作

取自主内存的像素首先被解包到适当数量的组成部分。OpenGL 的解包设备能够处理大量的不同格式的像素。接下来要利用像素图对数据进行缩放、移位，以及加工处理。根据数据的类型，操作结果被限制在适当的范围，然后或者将结果存于纹理内存用于纹理映射，或者被光栅化为片元。

如果像素数据是从帧缓存中读入的，那么还要执行像素变换操作（如缩放、移位、映射以及限值等）。像素变换的结果被打包成一种适当的格式，然后将其返回到处理器存储单元。

像素复制的操作类似于将解包操作与变换操作结合起来，除了打包和解包是不必要的以外，在数据写回帧缓存之前，通过变换操作只进行一次值的传递。

附 1.4　片元操作

如果纹理功能被激活，对于每一个片元在纹理内存中要产生一个纹理单元，并应用到片元上。如果雾被激活，还要进行雾的计算；之后，如果锯齿现象消除功能被激活，还要进行覆盖（锯齿现象的消除）计算。

接下来的操作就是剪取测试、alpha 测试（仅适用于 RGBA 模式）、模板测试、深度缓存测试以及抖动操作。所有的这些操作都可以禁止。下一步，如果是在索引模式下，如果已经指定了一种逻辑操作，就可以应用这一逻辑操作。如果在 RGBA 模式下，要执行混合操作。

根据不同的模式，要用颜色或索引掩码对片元进行屏蔽，然后将其画入适当的缓存。如果片元正被写入模板或深度缓冲器，那么屏蔽操作就要在模板测试和深度测试之后进行，然后勿需执行混合、抖动或逻辑操作，即可将结果画入帧缓存。

附 1.5　补充说明

矩阵操作用来处理当前矩阵堆栈，这些堆栈可以是模型视图矩阵堆栈、投影矩阵堆栈、或者纹理矩阵堆栈。命令 glMultMatrix*()，glLoadMatrix*()和 glLoadIdentity()应用于栈顶矩阵，而命令 glTranslate*()，glRotate*()，glScale*()，glOrtho()和 glFrustum()应用于创建与栈

顶矩阵相乘的矩阵，当模型视图矩阵被修改时，它的逆矩阵还用于法线向量变换。

设置当前光栅位置的命令严格地说，一直到进行光栅化操作，都是类似于顶点命令进行处理的。在光栅化操作时，光栅位置值被保存起来，并用于像素数据的光栅化操作。不同的glClear()命令将忽略除剪取、抖动、及写屏蔽以外的所有操作。

附录 2　OpenGL 状态变量

本附录列出可供查询的 OpenGL 状态变量、缺省值、以及获得这些状态变量值的命令。本附录还描述了 OpenGL 的出错处理技巧，以及怎样保存和恢复状态变量。OpenGL 参考手册中包含了本附录所描述的所有命令及常量的详细信息。本附录主要由以下主要内容组成：

- 查询命令。
- 出错处理。
- 状态变量的保存和恢复。
- OpenGL 的状态变量。

附 2.1　查询命令

状态变量可以利用下列四个命令简单地获得，同时决定指定的状态是否被激活。

---

void glGetBooleanv(GLenum pname, GLboolean *params);
void glGetIntegerv(GLenum pname, GLint *params);
void glGetFloatv(GLenum pname, GLfloat *params);
void glGetDoublev(GLenum pname, GLdouble *params);

这些命令用于获取布尔型、整型、浮点型或双精度型的状态变量。参数 pname 是一个表示返回状态变量的符号常量，参数 params 是指向放置返回数据的相应类型的数组的指针。pname 的可能值列于表“OpenGL 状态变量”中。如果需要返回期望的变量值，则必须进行状态变量的类型转换。

GLboolean glIsEnabled(GLenum cap)

如果 cap 指定的模式被激活，则函数返回值为 GL_TRUE；否则，返回 GL_FALSE。参数 cap 的可能值列于表“OpenGL 状态变量”中。

其他的特定命令返回特定的状态变量值。这里列出了这些命令的原型；可以利用表“OpenGL 状态变量”来查明何时需要使用这些命令。也可以参考《OpenGL 高性能图形库程序员参考手册》。在下一节将详细描述 OpenGL 的出错处理工具以及 glGetError()命令。

void glGetClipPlane(GLenum plane, GLdouble *equation);
GLenum glGetError(void);
void glGetLight{if}v(GLenum light, GLenum pname, TYPE *params);
void glGetMap{ifd}v(GLenum target, GLenum query, TYPE *v);
void glGetMaterial{if}v(GLenum face, GLenum pname, TYPE *params);
void glGetPixelMap{f ui us}v(GLenum map, TYPE *values);
void glGetPolygonStipple(GLubyte *mask);
const GLubyte * glGetString(GLenum name);
void glGetTexEnv{if}v(GLenum target, GLenum pname, TYPE *params);
void glGetTexGen{ifd}v(GLenum coord, GLenum pname, TYPE *params);
void glGetTexImage(GLenum target, GLint level, GLenum format,GLenum type, GLvoid *pixels);
void glGetTexLevelParameter{if}v(GLenum target, GLint level, GLenum pname, TYPE *params);
void glGetTexParameter{if}v(GLenum target, GLenum pname, TYPE *params);

附 2.2 出错处理

当 OpenGL 检测出一个错误时，它会记录下当前的错误代码。引起错误的命令在此时通常被忽略，因此该错误不会影响 OpenGL 的状态，也不会影响帧缓存中的内容。然而，如果记录的错误代码是 GL_OUT_OF_MEMORY，那么这条命令的结果是不确定的。一旦记录下当前错误代码，则该错误代码不会被清除，也就是说，不会加入错误代码记录。除非调用查询命令 glGetError()，才会得以返回当前错误代码。在分布式的 OpenGL 实现中，可能有多个当前错误代码，在查询之前，每一个错误代码都将保持原有设置。一旦读者已经查询了所有的当前错误代码，或者根本就没有错误，那么 glGetError()命令将返回 GL_NO_ERROR。因此，如果已经获得了一个错误代码，应该保持不断调用 glGetError()命令的习惯，直到返回 GL_NO_ERROR 为止。这样一来，就可以确保发现所有错误。附表 2-1 详细列出了 OpenGL 定义的错误代码。

可以使用 GLU 子函数 gluErrorString()来获得一个描述性的字符串，这个字符串与错误代码相对应。注意，一旦检测到错误，GLU 子函数经常返回一个错误值。GLU 定义的错误代码 GLU_INVALID_ENUM，GLU_INVALID_VALUE 和 GLU_OUT_OF_MEMORY 与相关的 OpenGL 代码具有相同的含义。

附表 2-1 OpenGL 出错代码

错误代码	描述性说明
GL_INVALID_ENUM	Glenum 参数超出范围
GL_INVALID_VALUE	Numeric 参数超出范围
GL_INVALID_OPERATION	当前声明无效
GL_STACK_OVERFLOW	堆栈上溢
GL_STACK_UNDERFLOW	堆栈下溢
GL_OUT_OF_MEMORY	内存不足

附 2.3 状态变量的保存和恢复

可以利用命令 glPushAttrib()和 glPopAttrib()保存和恢复属性堆栈的状态变量集的值。属性堆栈的深度至少为 16，可以利用参数 GL_MAX_ATTRIB_STACK_DEPTH 调用 glGetIntegerv()命令来获得实际的堆栈深度。数值为 GL_MAX_ATTRIB_STACK_DEPTH ，压入一个已满的堆栈或从一个空堆栈中弹出矩阵都会产生错误。

一般情况下，利用 glPushAttrib()和 glPopAttrib()命令比你自己获取和恢复这些状态变量值要快一些。有些值可能是以硬件方式压入和弹出的，保存和恢复它们可能需要更昂贵的开销。还有，如果是在远程客户机上操作，那么所有的属性数据都要经过网络进行传输，就像是在进行保存和恢复那样。然而，OpenGL 实现是将属性堆栈保存在服务器材上的，这样就避免了不必要的网络延迟。

---

void glPushAttrib(GLbitfield mask);

可以通过将 mask 中的位指定的所有属性压入属性堆栈中，来存储这些属性。附表 2-2 中列出了可能的屏蔽位，这些屏蔽位通过逻辑或（OR）连接，来保存属性的任意结合。每一位是与独立的状态变量集相对应的。比如，GL_LIGHTING_BIT 命令指所有的与光相关的状态变量的集合，它包括当前的材料颜色、环境光、散射、镜相、以及放射光，激活的光源列表，还有聚光灯的方向等。在调用 glPopAttrib()命令时，所有的这些变量都被恢复。要确切地为特殊的屏蔽值找出需要保存哪些属性，请参看“OpenGL 状态变量”表。

附表 2-2 属性列表

屏蔽位	属性组
GL_ACCUM_BUFFER_BIT	累积缓存
GL_ALL_ATTRIB_BITS	--
GL_COLOR_BUFFER_BIT	颜色缓存

续表

屏蔽位	属性组
GL_CURRENT_BIT	当前缓存
GL_DEPTH_BUFFER_BIT	深度缓存
GL_ENABLE_BIT	激活
GL_EVAL_BIT	求值程序
GL_FOG_BIT	雾
GL_HINT_BIT	提示
GL_LIGHTING_BIT	光照
GL_LINE_BIT	直线
GL_LIST_BIT	列表
GL_PIXEL_MODE_BIT	像素
GL_POINT_BIT	点
GL_POLYGON_BIT	多边形
GL_POLYGON_STIPPLE_BIT	多边形点划
GL_SCISSOR_BIT	剪取
GL_STENCIL_BUFFER_BIT	模板缓存
GL_TEXTURE_BIT	纹理缓存
GL_TRANSFORM_BIT	变换
GL_VIEWPORT_BIT	视区

void glPopAttrib(void);

恢复上一条 glPushAttrib()命令保存的状态变量的值。

附 2.4 OpenGL 的状态变量

下面是可供查询的状态变量列表，表中详细的列出了每一个状态变量，包括此变量的特征属性、初始值、最小值以及获得该状态变量建议使用的 glGet*()命令等等。可用 glGetBooleanv()，glGetIntegerv()，glGetFloatv()或者 glGetDoublev()命令获取的状态变量及命令都列于表中——这个命令最适合于给定的返回数据类型。有些状态变量不能通过 glIsEnabled()命令来获得。然而，作为查询命令，由 glIsEnabled()命令列出的状态变量也可以利用 glGetBooleanv()，glGetIntegerv()，glGetFloatv()和 glGetDoublev()等命令来获得。作为查询命令，由其他任意命令列出的状态变量则仅能以该命令获取。若没有列出属性组，那

么变量就不属于任何一组属性。除了与实现相关的变量外，所有可查询变量都有一个初始值。如果没有列出初始值，读者需要参考讨论该变量的章节（或者 OpenGL 参考手册），以便确定它的初始值。

附 2.4.1　当前值和相关数据

附表 2-3　当前值及与其相关的状态变量值

状态变量	描述	属性组	初始值	获取命令
GL_CURRENT_COLOR	当前颜色	当前属性	1, 1, 1, 1	glGetIntegerv(), glGetFloatv()
GL_CURRENT_INDEX	当前颜色索引	当前属性	1	glGetIntegerv() , glGetFloatv()
GL_CURRENT_TEXTURE_COORDS	当前纹理坐标	当前属性	0, 0, 0, 1	glGetFloatv()
GL_CURRENT_NORMAL	当前法线	当前属性	0, 0, 1	glGetFloatv()
GL_CURRENT_RASTER_POSITION	当前光栅位置	当前属性	0, 0, 0, 1	glGetFloatv()
GL_CURRENT_RASTER_DISTANCE	当前光栅距离	当前属性	0	glGetFloatv()
GL_CURRENT_RASTER_COLOR	与光栅位置相关的颜色	当前属性	1, 1, 1, 1	glGetIntegerv(), glGetFloatv()
GL_CURRENT_RASTER_INDEX	与光栅位置相关的颜色索引	当前属性	1	glGetIntegerv() , glGetFloatv()
GL_CURRENT_RASTER_TEXTURE_COORDS	与光栅位置相关的纹理坐标	当前属性	0, 0, 0, 1	glGetFloatv()
GL_CURRENT_RASTER_POSITION_ VALID	光栅位置的有效位	当前属性	GL_TRUE	glGetBooleanv()
GL_EDGE_FLAG	边缘标记	当前属性	GL_TRUE	glGetBooleanv()

附 2.4.2　变换

附表 2-4　状态变量的变换

状态变量	描述	属性组	初始值	获取命令
GL_MODELVIEW_MATRIX	模式视图矩阵堆栈	--	单位阵	glGetFloatv()
GL_PROJECTION_MATRIX	投影矩阵堆栈	--	单位阵	glGetFloatv()

续表

状态变量	描述	属性组	初始值	获取命令
GL_TEXTURE_MATRIX	纹理矩阵堆栈	--	单位阵	glGetFloatv()
GL_VIEWPORT	视区原点及范围	视区	—	glGetIntegerv()
GL_DEPTH_RANGE	深度的近、远范围	视区	0, 1	glGetFloatv()
GL_MODELVIEW_ STACK_DEPTH	模式视图矩阵堆栈指针	--	1	glGetIntegerv()
GL_PROJECTION_ STACK_DEPTH	投影矩阵堆栈指针	--	1	glGetIntegerv()
GL_TEXTURE_STACK_ DEPTH	纹理矩阵指针	--	1	glGetIntegerv()
GL_MATRIX_MODE	当前矩阵模式	变换	GL_MODELVI EW	glGetIntegerv()
GL_NORMALIZE	归一化法线开/关	变换/激活	GL_FALSE	glIsEnabled()
GL_CLIP_PLANEi	用户裁剪面系数	变换	0, 0, 0, 0	glGetClipPlane()

附 2.4.3 颜色

附表 2-5 颜色状态变量

状态变量	描述	属性组	初始值	获取命令
GL_FOG_COLOR	雾的颜色	雾	0, 0, 0, 0	glGetFloatv()
GL_FOG_INDEX	雾的索引	雾	0	glGetFloatv()
GL_FOG_DENSITY	指数变化雾的密度	雾	1.0	glGetFloatv()
GL_FOG_START	线性变化雾的开始	雾	0.0	glGetFloatv()
GL_FOG_END	线性变化雾的结束	雾	1.0	glGetFloatv()
GL_FOG_MODE	雾的模式	雾	GL_EXP	glGetIntegerv()
GL_FOG	如果雾激活，则为真	雾/激活	GL_FALSE	glIsEnabled()
GL_SHADE_MODEL	设置 glShadeModel()	光照	GL_SMOOTH	glGetIntegerv()

附 2.4.4 光照

初始值请参看表 7-1 和表 7-2。

附表 2-6 光照状态变量

状态变量	描述	属性组	初始值	获取命令
GL_LIGHTING	若光照激活，则为真	光照/激活	GL_FALSE	glIsEnabled()
GL_COLOR_MATERIAL	若颜色跟踪被激活，则为真	光照	GL_FALSE	glIsEnabled()

续表

状态变量	描述	属性组	初始值	获取命令
GL_COLOR_MATERIAL_PARAMETER	跟踪当前颜色的材料属性	光照	GL_AMBIENT_AND_DIFFUSE	glGetIntegerv()
GL_COLOR_MATERIAL_FACE	受颜色跟踪影响的表面	光照	GL_FRONT_AND_BACK	glGetIntegerv()
GL_AMBIENT	周围材料颜色	光照	(0.2,0.2,0.2,1.0)	glGetMaterialfv()
GL_DIFFUSE	漫反射材料颜色	光照	(0.8,0.8,0.8,1.0)	glGetMaterialfv()
GL_SPECULAR	镜面反射材料颜色	光照	(0.0,0.0,0.0,1.0)	glGetMaterialfv()
GL_EMISSION	放射材料颜色	光照	(0.0,0.0,0.0,1.0)	glGetMaterialfv()
GL_SHININESS	镜面反射指数	光照	0.0	glGetMaterialfv()
GL_LIGHT_MODEL_AMBIENT	周围场景颜色	光照	(0.2,0.2,0.2,1.0)	glGetFloatv()
GL_LIGHT_MODEL_LOCAL_VIEWER	局部观察者	光照	GL_FALSE	glGetBooleanv()
GL_LIGHT_MODEL_TWO_SIDE	使用双面光照	光照	GL_FALSE	glGetBooleanv()
GL_AMBIENT	灯光 i 的环境光强度	光照	(0.0,0.0,0.0,1.0)	glGetLightfv()
GL_DIFFUSE	灯光 i 的漫射强度	光照	-	glGetLightfv()
GL_SPECULAR	灯光 i 的镜面反射强度	光照	--	glGetLightfv()
GL_POSITION	灯光 i 的位置	光照	(0.0,0.0,1.0,0.0)	glGetLightfv()
GL_CONSTANT_ATTENUATION	常量衰减因子	光照	1.0	glGetLightfv()
GL_LINEAR_ATTENUATION	线性衰减因子	光照	0.0	glGetLightfv()
GL_QUADRATIC_ATTENUATION	平方衰减因子	光照	0.0	glGetLightfv()
GL_SPOT_DIRECTION	灯光 i 的聚光方向	光照	(0.0,0.0,-1.0)	glGetLightfv()
GL_SPOT_EXPONENT	灯光 i 的聚光指数	光照	0.0	glGetLightfv()
GL_SPOT_CUTOFF	灯光 i 的聚光角度	光照	180.0	glGetLightfv()
GL_LIGHTi	如果灯光 i 被激活，则为真	光照/激活	GL_FALSE	glIsEnabled()
GL_COLOR_INDEXES	ca, cd,和 cs 灯光颜色索引	光照/激活	0, 1, 1	glGetFloatv()

附 2.4.5 光栅化

附表 2-7 光栅化状态变量

状态变量	描述	属性组	初始值	使用命令
GL_POINT_SIZE	点的尺寸	点	1.0	glGetFloatv()
GL_POINT_SMOOTH	消除点的锯齿现象	点/激活	GL_FALSE	glIsEnabled()
GL_LINE_WIDTH	线宽	线	1.0	glGetFloatv()
GL_LINE_SMOOTH	消除线的锯齿现象	线/激活	GL_FALSE	glIsEnabled()
GL_LINE_STIPPLE_PATTERN	直线点划	线	1's	glGetIntegerv()
GL_LINE_STIPPLE_REPEAT	重复直线点划	线	1	glGetIntegerv()
GL_LINE_STIPPLE	直线点划激活	线/激活	GL_FALSE	glIsEnabled()
GL_CULL_FACE	多边形选择激活	多边形/激活	GL_FALSE	glIsEnabled()
GL_CULL_FACE_MODE	选择正/背面多边形	多边形	GL_BACK	glGetIntegerv()
GL_FRONT_FACE	多边形正面 cw/ccw 指示器	多边形	GL_CCW	glGetIntegerv()
GL_POLYGON_SMOOTH	消除多边形锯齿现象	多边形/激活	GL_FALSE	glIsEnabled()
GL_POLYGON_MODE	多边形光栅化模式	多边形	GL_FILL	glGetIntegerv()
GL_POLYGON_STIPPLE	激活多边形点划	多边形/激活	GL_FALSE	glIsEnabled()
--	多边形点划模式	多边形点划	1's	glGetPolygon-Stipple()

附 2.4.6 纹理

附表 2-8 纹理状态变量

状态变量	描述	属性组	初始值	使用命令
GL_TEXTURE_x	若 x 维纹理激活（x 为一维或二维）	真/激活	GL_FALSE	glIsEnabled()
GL_TEXTURE	详细等级为 i 的 x 维纹理图像	--	--	glGetTexImage()
GL_TEXTURE_WIDTH	x 维纹理图像 i 的宽度	--	0	glGetTexLevelParameter()

续表

状态变量	描述	属性组	初始值	使用命令
GL_TEXTURE_HEIGHT	x 维纹理图像 i 的高度	--	0	glGetTexLevelParameter()
GL_TEXTURE_BORDER	x 维纹理图的边界宽度	--	0	glGetTexLevelParameter()
GL_TEXTURE_COMPONENTS	纹理图形组成	--	1	glGetTexLevelParameter()
GL_TEXTURE_BORDER_COLOR	纹理边界颜色	纹理	0, 0, 0, 0	glGetTexParameter()
GL_TEXTURE_MIN_FILTER	纹理缩小函数	纹理	GL_NEAREST_MIPMAP_LINEAR	glGetTexParameter()
GL_TEXTURE_MAG_FILTER	纹理扩大函数	纹理	GL_LINEAR	glGetTexParameter()
GL_TEXTURE_WRAP_x	纹理覆盖模式（x 是 S 或 T）	纹理	GL_REPEAT	glGetTexParameter()
GL_TEXTURE_ENV_MODE	纹理应用函数	纹理	GL_MODULATE	glGetTexEnviv()
GL_TEXTURE_ENV_COLOR	纹理环境函数	纹理	0, 0, 0, 0	glGetTexEnvfv()
GL_TEXTURE_GEN_x	纹理情报激活（x 是 S，T，R，或 Q）	纹理/激活	GL_FALSE	glIsEnabled()
GL_EYE_LINEAR	纹理情报平面方程系数	纹理	--	glGetTexGenfv()
GL_OBJECT_LINEAR	纹理物体线性系数	纹理	--	glGetTexGenfv()
GL_TEXTURE_GEN_MODE	纹理情报使用函数	纹理	GL_EYE_LINEAR	glGetTexGeniv()

附 2.4.7　像素操作

附表 2-9　像素操作

状态变量	描述	属性组	初始值	使用命令
GL_SCISSOR_TEST	剪切激活	剪切/激活	GL_FALSE	glIsEnabled()
GL_SCISSOR_BOX	剪切框	剪切	--	glGetIntegerv()
GL_STENCIL_TEST	模板激活	模板缓存/激活	GL_FALSE	glIsEnabled()
GL_STENCIL_FUNC	模板函数	模板缓存	GL_ALWAYS	glGetIntegerv()
GL_STENCIL_VALUE_MASK	模板掩码	模板缓存	1's	glGetIntegerv()

续表

状态变量	描述	属性组	初始值	使用命令
GL_STENCIL_REF	模板参考值	模板缓存	0	glGetIntegerv()
GL_STENCIL_FAIL	模板失败动作	模板缓存	GL_KEEP	glGetIntegerv()
GL_STENCIL_PASS_DEPTH_FAIL	模板深度缓存失败动作	模板缓存	GL_KEEP	glGetIntegerv()
GL_STENCIL_PASS_DEPTH_PASS	模板深度缓存传递动作	模板缓存	GL_KEEP	glGetIntegerv()
GL_ALPHA_TEST	Alpha 测试激活	模板缓存 /激活	GL_FALSE	glIsEnabled()
GL_ALPHA_TEST_FUNC	Alpha 测试函数	颜色缓存	GL_ALWAYS	glGetIntegerv()
GL_ALPHA_TEST_REF	Alpha 测试参考值	颜色缓存	0	glGetIntegerv()
GL_DEPTH_TEST	深度缓存激活	深度缓存/激活	GL_FALSE	glIsEnabled()
GL_DEPTH_FUNC	深度缓存测试函数	深度缓存	GL_LESS	glGetIntegerv()
GL_BLEND	混合激活	颜色缓存/激活	GL_FALSE	glIsEnabled()
GL_BLEND_SRC	混合源函数	颜色缓存	GL_ONE	glGetIntegerv()
GL_BLEND_DST	混合目的文件函数	颜色缓存	GL_ZERO	glGetIntegerv()
GL_LOGIC_OP	激活逻辑操作	颜色缓存/激活	GL_FALSE	glIsEnabled()
GL_LOGIC_OP_MODE	逻辑操作函数	颜色缓存	GL_COPY	glGetIntegerv()
GL_DITHER	激活抖动	颜色缓存/激活	GL_TRUE	glIsEnabled()

附 2.4.8 帧缓存控制

附表 2-10 帧缓存控制状态变量

状态变量	描述	属性组	初始值	使用命令
GL_DRAW_BUFFER	选择绘制缓存	颜色缓存	--	glGetIntegerv()
GL_INDEX_WRITEMASK	写掩码颜色索引	颜色缓存	1's	glGetIntegerv()
GL_COLOR_WRITEMASK	写颜色激活 R,G,B,或 A	颜色缓存	GL_TRUE	glGetBooleanv()
GL_DEPTH_WRITEMASK	为写入而激活的深度缓存	深度缓存	GL_TRUE	glGetBooleanv()
GL_STENCIL_WRITEMASK	写入掩码的模板缓存	模板缓存	1's	glGetIntegerv()
GL_COLOR_CLEAR_VALUE	颜色缓存清除值（RGBA 模式）	颜色缓存	0, 0, 0, 0	glGetFloatv()
GL_INDEX_CLEAR_VALUE	索引缓存清除值（颜色索引模式）	颜色缓存	0	glGetFloatv()
GL_DEPTH_CLEAR_VALUE	深度缓存清除值	深度缓存	1	glGetIntegerv()
GL_STENCIL_CLEAR_VALUE	模板缓存清除值	模板缓存	0	glGetIntegerv()
GL_ACCUM_CLEAR_VALUE	累积缓存清除值	累积缓存	0	glGetFloatv()

附 2.4.9　像素

附表 2-11　像素状态变量

状态变量	描述	属性组	初始值	使用命令
GL_UNPACK_SWAP_BYTES	GL_UNPACK_SWAP_BYTES 的值	--	GL_FALSE	glGetBooleanv()
GL_UNPACK_LSB_FIRST	GL_UNPACK_LSB_FIRST 的值	--	GL_FALSE	glGetBooleanv()
GL_UNPACK_ROW_LENGTH	GL_UNPACK_ROW_LENGTH 的值	--	0	glGetIntegerv()
GL_UNPACK_SKIP_ROWS	GL_UNPACK_SKIP_ROWS 的值	--	0	glGetIntegerv()
GL_UNPACK_SKIP_PIXELS	GL_UNPACK_SKIP_PIXELS 的值	--	0	glGetIntegerv()
GL_UNPACK_ALIGNMENT	GL_UNPACK_ALIGNMENT 的值	--	4	glGetIntegerv()
GL_PACK_SWAP_BYTES	GL_PACK_SWAP_BYTES 的值	--	GL_FALSE	glGetBooleanv()
GL_PACK_LSB_FIRST	GL_PACK_LSB_FIRST 的值	--	GL_FALSE	glGetBooleanv()
GL_PACK_ROW_LENGTH	GL_PACK_ROW_LENGTH 的值	--	0	glGetIntegerv()
GL_PACK_SKIP_ROWS	GL_PACK_SKIP_ROWS 的值	--	0	glGetIntegerv()
GL_PACK_SKIP_PIXELS	GL_PACK_SKIP_PIXELS 的值	--	0	glGetIntegerv()
GL_PACK_ALIGNMENT	GL_PACK_ALIGNMENT 的值	--	4	glGetIntegerv()
GL_MAP_COLOR	若颜色进行映射则为真	像素	GL_FALSE	glGetBooleanv()
GL_MAP_STENCIL	若模板值被映射则为真	像素	GL_FALSE	glGetBooleanv()
GL_INDEX_SHIFT	GL_INDEX_SHIFT 的值	像素	0	glGetIntegerv()
GL_INDEX_OFFSET	GL_INDEX_OFFSET 的值	像素	0	glGetIntegerv()
GL_x_SCALE	GL_x_SCALE 的值; x 是 GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, 或 GL_DEPTH	像素	1	glGetFloatv()
GL_x_BIAS	Value of GL_x_BIAS; x 是 GL_RED, GL_GREEN, GL_BLUE, GL_ ALPHA 或 GL_DEPTH 中之一	像素	0	glGetFloatv()
GL_ZOOM_X	x 图形放大因子	像素	1.0	glGetFloatv()
GL_ZOOM_Y	y 图形放大因子	像素	1.0	glGetFloatv()
GL_x	glPixelMap() 变换表	像素	0's	glGetPixelMap()
GL_x_SIZE	表 x 的尺寸	像素	1	glGetIntegerv()
GL_READ_BUFFER	读源缓存	像素	--	glGetIntegerv()

附 2.4.10 求值程序

附表 2-12 求值程序的状态变量

状态变量	描述	属性组	初始值	使用命令
GL_ORDER	1D 映射顺序	--	1	glGetMapiv()
GL_ORDER	2D 映射顺序	--	1, 1	glGetMapiv()
GL_COEFF	1D 控制点	--	--	glGetMapfv()
GL_COEFF	2D 控制点	--	--	glGetMapfv()
GL_DOMAIN	1D 区结束点	--	--	glGetMapfv()
GL_DOMAIN	2D 区结束点	--	--	glGetMapfv()
GL_MAP1_x	1D 图激活: x 是图类型	估值/激活	GL_FALSE	glIsEnabled()
GL_MAP2_x	2D 图激活: x 是图类型	估值/激活	GL_FALSE	glIsEnabled()
GL_MAP1_GRID_DOMAIN	1D 网格端点	估值	0, 1	glGetFloatv()
GL_MAP2_GRID_DOMAIN	2D 网格端点	估值	0, 1; 0, 1	glGetFloatv()
GL_MAP1_GRID_SEGMENTS	1D 网格分界	估值	1	glGetFloatv()
GL_MAP2_GRID_SEGMENTS	2D 网格分界	估值	1,1	glGetFloatv()
GL_AUTO_NORMAL	若自动标准的发生被激活	估值	GL_FALSE	glIsEnabled()

附 2.4.11 提示

附表 2-13 提示状态变量

状态变量	描述	属性组	初始值	使用命令
GL_PERSPECTIVE_ CORRECTION_ HINT	透视图修正提示	提示	GL_DONT_CARE	glGetIntegerv()
GL_POINT_SMOOTH_HINT	点平滑提示	提示	GL_DONT_CARE	glGetIntegerv()
GL_LINE_SMOOTH_HINT	直线平滑提示	提示	GL_DONT_CARE	glGetIntegerv()
GL_POLYGON_SMOOTH_ HINT	多边形的平滑提示	提示	GL_DONT_CARE	glGetIntegerv()
GL_FOG_HINT	雾的提示	提示	GL_DONT_CARE	glGetIntegerv()

附 2.4.12 依赖执行的值

附表 2-14 随执行而定的状态变量

状态变量	描述	属性组	初始值	使用命令
GL_MAX_LIGHTS	光的最大数量	--	8	glGetIntegerv()
GL_MAX_CLIP_PLANES	用户剪切平面的最大数量	--	6	glGetIntegerv()

续表

状态变量	描述	属性组	初始值	使用命令
GL_MAX_MODELVIEW_STACK_DEPTH	模型视图矩阵堆栈深度最大值	--	32	glGetIntegerv()
GL_MAX_PROJECTION_STACK_DEPTH	投影矩阵深度最大值	--	2	glGetIntegerv()
GL_MAX_TEXTURE_STACK_DEPTH	纹理矩阵深度最大值	--	2	glGetIntegerv()
GL_SUBPIXEL_BITS	在 x 和 y 子像素精度位数值	--	4	glGetIntegerv()
GL_MAX_TEXTURE_SIZE	纹理图像的最大高度或宽度（w/o 边界）	--	64	glGetIntegerv()
GL_MAX_PIXEL_MAP_TABLE	glPixelMap()变换表的最大尺寸	--	32	glGetIntegerv()
GL_MAX_NAME_STACK_DEPTH	选择命名堆栈的深度最大值	--	64	glGetIntegerv()
GL_MAX_LIST_NESTING	调用嵌套显示表最大值	--	64	glGetIntegerv()
GL_MAX_EVAL_ORDER	求值多项式次数的最大值	--	8	glGetIntegerv()
GL_MAX_VIEWPORT_DIMS	视区尺寸的最大值	—	—	glGetIntegerv()
GL_MAX_ATTRIB_STACK_DEPTH	特征堆栈深度的最大值	--	16	glGetIntegerv()
GL_AUX_BUFFERS	辅助缓存的数量	--	0	glGetBooleanv()
GL_RGBA_MODE	若颜色缓存存储 RGBA 则值为真	--	--	glGetBooleanv()
GL_INDEX_MODE	若颜色缓存存储索引则值为真	--	--	glGetBooleanv()
GL_DOUBLEBUFFER	若前&后缓存存在则值为真	--	--	glGetBooleanv()
GL_STEREO	若左&右缓存存在则值为真	--	--	glGetFloatv()
GL_POINT_SIZE_RANGE	消除锯齿现象点的尺寸范围（由低到高）	--	1, 1	glGetFloatv()
GL_POINT_SIZE_GRANULARITY	消除锯齿现象点尺寸的粒度	--	--	glGetFloatv()
GL_LINE_WIDTH_RANGE	消除锯齿现象直线的宽度范围（由低到高）	--	1, 1	glGetFloatv()
GL_LINE_WIDTH_GRANULARITY	消除锯齿现象的宽度直线的粒度	--	--	glGetFloatv()

附 2.4.13 像素深度

附表 2-15 像素深度状态变量

状态变量	描述	属性组	初始值	使用命令
GL_RED_BITS	在颜色缓存每一红色分量的位数	--	--	glGetIntegerv()
GL_GREEN_BITS	在颜色缓存每一绿色分量的位数	--	--	glGetIntegerv()
GL_BLUE_BITS	在颜色缓存每一蓝色分量的位数	--	--	glGetIntegerv()
GL_ALPHA_BITS	在颜色缓存每一 alpha 分量的位数	--	--	glGetIntegerv()
GL_INDEX_BITS	在颜色缓存每一索引的位数	--	--	glGetIntegerv()
GL_DEPTH_BITS	深度缓存的位平面数量	--	--	glGetIntegerv()
GL_STENCIL_BITS	模板位平面数量	--	--	glGetIntegerv()
GL_ACCUM_RED_BITS	在累积缓存每一红色分量的位数	--	--	glGetIntegerv()
GL_ACCUM_GREEN_BITS	在累积缓存每一绿色分量的位数	--	--	glGetIntegerv()
GL_ACCUM_BLUE_BITS	在累积缓存每一蓝色分量的位数	--	--	glGetIntegerv()
GL_ACCUM_ALPHA_BITS	在颜色缓存每一 alpha 分量的位数	--	--	glGetIntegerv()

附 2.4.14 混合

附表 2-16 混合状态变量

状态变量	描述	属性组	初始值	使用命令
GL_LIST_BASE	glListBase()的设置	列表	0	glGetIntegerv()
GL_LIST_INDEX	构造的显示列表的数目：若无为 0	--	0	glGetIntegerv()
GL_LIST_MODE	构造的显示列表的模式：若无为未定义	--	0	glGetIntegerv()
GL_ATTRIB_STACK_DEPTH	特征堆栈指示器	--	0	glGetIntegerv()
GL_NAME_STACK_DEPTH	命名堆栈深度	--	0	glGetIntegerv()
GL_RENDER_MODE	设置 glRenderMode()命令的当前错误代码	--	GL_RENDER	glGetIntegerv()
--	当前错误代码	--	0	glGetError()

附录 3　OpenGL 实用库

OpenGL 提供了一系列强大而精练的绘图操作，所有高级绘图操作必须以这些形式完成。为了简化编程工作，OpenGL 实用库（GLU）包括了封装 OpenGL 命令的实用子程序。在正文中相关章节已经对其中的多数程序进行过描述。为完整起见，在这里将它们简要地列出来。在前面没有提到的 GLU 子程序在此进行较详尽的描述。但更详细的描述请参见《OpenGL 参考手册》。本附录按功能将 GLU 子程序分成如下几组：

- 纹理映射中的图像操作。
- 变换坐标。
- 多边形网格化。
- 渲染球体、圆柱和圆盘。
- NURBS 曲面和曲线。
- 错误描述。

附 3.1　纹理映射中的图像操作

当应用程序中设置纹理映射时，可能要用到 mipmapping 技术，该技术需要一系列的简化的图像（或纹理图）。为了支持 mipmapping，GLU 包含了一个通用的子程序对图像进行缩放（gluScaleImage()），以及其他用于生成完整的 mipmaps 的子程序（gluBuild1DMipmaps() 和 gluBuild2DMipmaps()）。这些子程序已经进行过详细的讨论，因此这里只列出了它们的原型。

GLint gluScaleImage(GLenum format, GLint widthin, GLint heightin, GLenum typein,
const void *datain, GLint widthout, GLint heightout, GLenum typeout,
void *dataout);
GLint gluBuild1DMipmaps(GLenum target, GLint components, GLint width,
GLenum format, GLenum type, void *data);
GLint gluBuild2DMipmaps(GLenum target, GLint components, GLint width,
GLint height, GLenum format, GLenum type, void *data);

附 3.2 变换坐标

GLU 中包含为标准投影和正交视图创建矩阵的子程序（gluPerspective()及gluOrtho2D()）。另外，还有一个视图子程序用于设置观察点（gluLookAt()）。GLU 还包括了一个用于创建拾取矩阵的子程序（gluPickMatrix()），下面是四个子程序的原型。

---

void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);
void gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);
void gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble height, GLint viewport[4]);

另外，GLU 提供了两个子程序 gluProject()和 gluUnProject()，用于在物体坐标和屏幕坐标之间转换：

GLint gluProject(GLdouble objx, GLdouble objy, GLdouble objz, const GLdouble modelMatrix[16],const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *winx, GLdouble *winy, GLdouble *winz);

用 modelMatrix, projMatrix 和 viewport 将指定的物体坐标 objx, objy 和 objz 转换到窗口坐标下，其结果存储在 winx, winy, 和 winz 中。如果成功，返回 GL_TRUE，否则返回 GL_FALSE。

GLint gluUnProject(GLdouble winx, GLdouble winy, GLdouble winz, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *objx, GLdouble *objy, GLdouble *objz);

用 modelMatrix，projMatrix 和 viewport 将指定的窗口坐标 winx，winy，及 winz 转换到物体坐标下，其结果存储在 objx，objy 和 objz 中。如果成功返回 GL_TRUE，否则返回 GL_FALSE。

附 3.3　多边形网格化

正如“描述点、线和多边形”中讨论的那样，OpenGL 可以直接显示简单凸多边形。如果多边形的边只在顶点处相交，而且只有两条边在顶点处相交，则这个多边形是简单多边形。如果应用程序需要显示简单的非凸多边形，或是需要显示带有洞的简单多边形（类似于在一个大的简单多边形之中挖去一个相对较小的简单多边形所形成的图形），这些多边形在显示之前必须分割为凸多边形，这种分割称为网格化。GLU 提供了执行网格化操作的一组子程序。注意 GLU 网格化子程序不能处理非简单的多边形，对这样的多边形 OpenGL 没有提供标准的方法。

由于常常需要进行网格化操作，而且要求非常灵活地运用，因此本节详细描述 GLU 的网格化子程序。这些程序的输入是任意的、可以包括有孔洞的简单多边形，返回值为三角形、三角形网格和三角形扇面或者各类型的组合。如果不想处理网格和扇面，那么可以只用三角形。当然，如果关心程序执行性能的话，就应该利用任何可能获得的网格和扇面信息。

附 3.4　回调机制

要用 GLU 对多边形进行网格化，首先需要创建一个网格化的物体，然后在网格化过程中的适当的时间提供需调用的回调函数。指定回调函数之后，利用 GLU 子程序来描述多边形及孔洞，这些操作类似于 OpenGL 的多边形子程序。描述完多边形以后，网格化工具将需要的回调函数激活。

回调函数通常用于保存三角形、三角形网格和三角形扇面中由用户定义的数据，或以 OpenGL 显示列表形式存储数据。为了渲染多边形，其他代码用于转换数据结构，或调用显示列表。尽管回调函数可以调用 OpenGL 命令来直接显示它们，但通常并不这么做，因为网格化的计算量是非常大的。如果要再次显示该图形的话，最好的办法是将数据存储起来。GLU 网格化子程序不会返回任何新的顶点，因此不必对顶点、纹理坐标或颜色进行插值。

附 3.5　网格化物体

在描述和网格化复杂多边形时，多边形数据，如顶点及边是和回调函数联系在一起的。为了实现网格化，应用程序首先必须应用函数 gluNewTess()创建一个网格化物体。

GLUtriangulatorObj* gluNewTess(void);

创建新的网格化物体并返回指向该物体的指针，如果创建失败则返回一个空的指针。

如果不再需要网格化物体，可以利用函数 gluDeleteTess()将其删除并释放所有相关的内存。

---

void gluDeleteTess(GLUtriangulatorObj *tessobj);

删除指定的网格化物体 tessobj，并释放所有相关的内存。一个网格化物体可以在所有的网格化操作中反复应用。

附 3.6 指定回调函数

一个网格化物体最多可以指定 5 个回调函数。回调函数由函数 gluTessCallback()来指定。

---

void gluTessCallback(GLUtriangulatorObj *tessobj, GLenum type, void (*fn)());

将回调函数 fn 与网格化物体 tessobj 联系在一起。回调的类型由参数 type 决定，type 可以为 GLU_BEGIN, GLU_EDGE_FLAG, GLU_VERTEX, GLU_END, 或 GLU_ERROR 中的任一个。5 种可能的回调函数的原型如下：

GLU_BEGIN	void begin(GLenum type);
GLU_EDGE_FLAG	void edgeFlag(GLboolean flag);
GLU_VERTEX	void vertex(void *data);
GLU_END	void end(void);
GLU_ERROR	void error(GLenum error);

要改变回调函数，只需以新的函数名调用 gluTessCallback()命令即可。如果要删除一个回调函数，只需将适当的函数空指针传给 gluTessCallback()函数即可。

在网格化过程中，这些函数均是与 OpenGL 命令 glBegin()，glEdgeFlag*()，glVertex*()以及 glEnd()类似的形式调用的。如果在网格化中出现错误，就会激活回调函数 error()。

GLU_BEGIN 回调以三个可能的参数之一来激发：三个参数为 GL_TRIANGLE_FAN，GL_TRIANGLE_STRIP 或 GL_TRIANGLES。在调用该函数后，并在与 GLU_END 相关的回调发生之前，GLU_EDGE_FLAG 和 GLU_VERTEX 回调将被激活。相关的顶点和边缘标记和它们在 OpenGL 中 glBegin(GL_TRIANGLE_FAN)，glBegin (GL_TRIANGLE_STRIP)或 glBegin(GL_TRIANGLES)与其相对应的 glEnd()之间的标记的解释是相同的。

回调函数 error 传递的是 GLU 错误值。用函数 gluErrorString()可以获得一个描述错误的字符串。关于这个函数的详细信息，请参见“错误的描述”一节。

附 3.7 描述网格化多边形

被网格化的多边形，可能带有洞，由下列四个函数来定义：gluBeginPolygon()，gluTessVertex()，gluNextContour()和 gluEndPolygon()。对于没有孔洞的多边形，此定义与 OpenGL 中一样：以 gluBeginPolygon()开始，在边界上的每一个顶点调用 gluEndPolygon()，最后以调用 gluEndPolygon()结束这个多边形。如果一个多边形由数个轮廓线组成（包括孔洞和带有洞中洞的多边形），轮廓线要一个接一个的定义，每一个轮廓线定义前面有一个 gluNextContour()命令。当调用 gluEndPolygon()时，表示完成了最后一个轮廓线的定义，开始网格化。可以在第一条轮廓线之前忽略对 gluNextContour()命令的调用。下面是这些函数的详细描述。

---

void gluBeginPolygon(GLUtriangulatorObj *tessobj);

开始定义要网格化的多边形，并与网格化物体 tessobj 相联系。

---

void gluTessVertex(GLUtriangulatorObj *tessobj,GLdouble v[3], void * data);

指定网格化多边形的顶点。在网格化多边形中，每一个顶点都要调用该函数。tessobj 是要用到的网格化物体，v 包括了三维顶点的坐标，data 是一个任意的指针，被传回与 GLU_VERTEX 相关的回调函数。通常，它包含顶点数据，纹理坐标，颜色信息，或其他应用程序所需的信息。

---

void gluNextContour(GLUtriangulatorObj *tessobj, GLenum type);

当需网格化的多边形是由多个轮廓线组成时，该函数标记出下一条轮廓线的开始。参数 type 可以是 GLU_EXTERIOR，GLU_INTERIOR，GLU_CCW，GLU_CW，或 GLU_UNKNOWN。如果能够正确获取它们，网格化可能进行得更快。对于一个带有孔洞的多边形，有一条外轮廓线，其他为内轮廓线。gluNextContour()可以在 gluBeginPolygon()函数之后直接调用，但是如果没有直接调用，第一条轮廓线被假定为 GLU_EXTERIOR 类型。GLU_CW 和 GLU_CCW 表示顺时针和逆时针方向的多边形。在三维条件下顺时针和逆时针是任意的；但是在任意平面内，有两个的不同方向，使用时 GLU_CW 和 GLU_CCW 应该一致。如果你没有线索的话，就使用 GLU_UNKNOWN 值。

---

void gluEndPolygon(GLUtriangulatorObj *tessobj);

表示多边形定义的结束，紧接着就可以使用网格化物体 tessobj 进行网格化操作。

附 3.8 渲染球体、圆柱和圆盘

GLU 包括一系列以不同风格和方向绘制不同简单表面（球体、圆柱、盘和盘的一部分）的子程序。在 OpenGL 参考手册中对这些函数进行了详细描述。下面列出了它们的原型。

要创建二次曲面，可以使用 gluNewQuadric() 命令（在完成任务后，可用函数 gluDeleteQuadric()将其删除）。然后用适当的子程序来指定期望的渲染模式，操作如下所示。

- 确定是否生成表面法线，如果要生成，确定是每个顶点有一条法线还是每个表面有法线，所使用的函数为 gluQuadricNormals()。
- 确定是否生成纹理坐标，所使用的函数为 gluQuadricTexture()。
- 确定二次曲面的外侧及内侧，所使用的命令为 gluQuadricOrientation()。
- 确定二次曲面是以一系列多边形、线还是点来绘制，所使用的函数为 gluQuadricDrawStyle()。

在确定了渲染模式之后，还要简单地为期望的二次曲面类型激活渲染函数：gluSphere()，gluCylinder()，gluDisk()，或者 gluPartialDisk()。如果在渲染过程中发生错误，由 gluQuadricCallBack()命令指定的错误处理函数将被激活。

用*Radius，height，以及类似的参数来对二次曲面进行比例缩放要比使用 glScale*()命令更好，因此生成的单位法线不必进行反归一化。

函数原型有三种类型。

管理二次曲面物体：

GLUquadricObj* gluNewQuadric (void);
void gluDeleteQuadric (GLUquadricObj *state);
void gluQuadricCallback (GLUquadricObj *qobj, GLenum which, void (* fn)());

控制渲染：

---

void gluQuadricNormals (GLUquadricObj *quadObject, GLenum normals);
void gluQuadricTexture (GLUquadricObj *quadObject, GLboolean textureCoords);
void gluQuadricOrientation (GLUquadricObj *quadObject, GLenum orientation);
void gluQuadricDrawStyle (GLUquadricObj *quadObject, GLenum drawStyle);

定义二次曲面原语：

```
void gluCylinder (GLUquadricObj *qobj, GLdouble baseRadius, GLdouble topRadius,
    GLdouble height, GLint slices, GLint stacks);
void gluDisk (GLUquadricObj *qobj, GLdouble innerRadius, GLdouble outerRadius,
    GLint    slices, GLint loops);
void gluPartialDisk (GLUquadricObj *qobj, GLdouble innerRadius,
    GLdouble outerRadius,   GLint slices, GLint loops, GLdouble startAngle,
    GLdouble sweepAngle);
void gluSphere (GLUquadricObj *qobj, GLdouble radius, GLint slices ,GLint stacks);
```

附 3.9 NURBS 曲线和曲面

NURBS 子程序提供了对曲面和曲线的二维和三维的强大的描述。在许多计算机辅助机械设计系统中，它们被用于表达几何体。GLU 的 NURBS 子程序可以各种不同的风格渲染这类曲线和曲面，并且能够自动处理自适应分割，可以在高曲度和近侧面轮廓中将整个区域分割为更小的三角形。下面列出了这些函数的原型。

管理 NURBS 物体：

```
GLUnurbsObj* gluNewNurbsRenderer (void);
void gluDeleteNurbsRenderer (GLUnurbsObj *nobj);
void gluNurbsCallback (GLUnurbsObj *nobj, GLenum which, void (*fn )());
```

创建 NURBS 曲线：

```
void gluBeginCurve (GLUnurbsObj *nobj);
void gluEndCurve (GLUnurbsObj *nobj);
void gluNurbsCurve (GLUnurbsObj *nobj, GLint nknots, GLfloat *knot, GLint stride,
    GLfloat   *ctlarray, GLint order, GLenum type);
```

创建 NURBS 曲面：

```
void gluBeginSurface (GLUnurbsObj *nobj);
void gluEndSurface (GLUnurbsObj *nobj);
void gluNurbsSurface (GLUnurbsObj *nobj, GLint uknot_count, Glfloat *uknot, GLint
```

vknot_count, GLfloat *vknot, GLint u_stride, GLint v_stride, GLfloat *ctlarray, GLint uorder, GLint vorder, GLenum type);

定义修整区域：

---

void gluBeginTrim (GLUnurbsObj *nobj);
void gluEndTrim (GLUnurbsObj *nobj);
void gluPwlCurve (GLUnurbsObj *nobj, GLint count, GLfloat *array, GLint stride, GLenum type);

控制 NURBS 渲染：

---

void gluLoadSamplingMatrices (GLUnurbsObj *nobj, const GLfloat modelMatrix[16], const GLfloat projMatrix[16], const GLint viewport[4]);
void gluNurbsProperty (GLUnurbsObj *nobj, GLenum property, GLfloat value);
void gluGetNurbsProperty (GLUnurbsObj *nobj, GLenum property, GLfloat *value);

附 3.10 错误的描述

GLU 提供了用于获得描述错误代码的子程序。关于 OpenGL 错误处理的详细内容，请参见“错误的处理”一节。

---

const GLubyte* gluErrorString(GLenum errorCode);

返回一个指向描述字符串的指针，该字符串对应于 OpenGL、GLU 或是 GLX 在 errorCode 的错误号。OpenGL 定义的错误代码在参考手册中有详细的描述。

附录 4　OpenGL 对 X 窗口系统的扩展

本附录简单地讨论了作为 OpenGL 对 X 窗口系统的扩展（GLX）部分的子程序。这些子程序将在 OpenGL 参考手册中进行详细的讨论。如果有关于 X 系统的知识，将有助于理解下面的内容并很好地使用 GLX。本附录主要内容：

- 初始化。
- 控制渲染。
- GLX 原型。

在 X 窗口系统中，OpenGL 渲染是以标准的 X 意义扩展到 X 中的：用一般的 X 机制来实现连接。如同其他的 X 扩展一样，这里存在一种将 OpenGL 渲染命令封装到 X 字节流中的事先定义的网络协议。

附 4.1　初始化

OpenGL 用 glXQueryExtension()命令和 glXQueryVersion()命令来确定是否为 X 服务器定义了 GLX 扩展，如果定义了这种扩展，还需确定是哪一个版本。glXChooseVisual()子程序返回一个指向 XvisualInfo 结构的指针。可以用 glXGetConfig()查询特定的 OpenGL 实现所支持的视频属性。

附 4.2　控制渲染

OpenGL 提供了几个 GLX 子程序用于创建和管理 OpenGL 渲染内容。可以利用这些内容进行后台渲染。同时还提供了用于诸如在 X 和 OpenGL 流中同步执行任务、交换前后缓存、使用 X 字体等任务的子程序。

附 4.2.1　管理 OpenGL 渲染内容

OpenGL 渲染内容表是由 glXCreateContext()命令创建的。该函数参数之一允许程序员绕过 X 服务器直接请求渲染内容。注意，如果直接进行渲染，那么 X 服务器连接必须是本地的，并且 OpenGL 实现必须支持直接渲染。利用 glXIsDirect()命令决定 GLX 内容表是否可以直接渲染。

函数 glXMakeCurrent()用于制作当前渲染内容表；函数 glXGetCurrentContext()返回当前内容表。还可以使用函数 glXGetCurrentDrawable()来获得当前可画物。记住，对任何线程来

说，一次只能有一个内容表。如果你有多个内容表，可以用函数 glXDestroyContext()将选定的 OpenGL 状态变量从一个内容表中复制到另一个内容表。

附 4.2.2 后台渲染

为了进行后台渲染，需要首先创建一幅 X 位图，然后将它作为 glXCreateGLXPixmap()函数的一个参数。一旦渲染完成，即可利用函数 glXDestroyGLXPixmap()命令删除 X 和 GLX 位图之间的联系。

附 4.2.3 同步执行

在 OpenGL 渲染结束之前要避免 X 的执行请求，这可以利用函数 glXWaitGL()。这样就可以确保先前发出的 OpenGL 命令在 X 渲染调用之前执行。尽管利用函数 glFinish()也可以得到相同的结果，但是 glXWaitGL()命令却不需要绕过 X 服务器，因此其效率更高。

附 4.2.4 交换缓存

对于双缓存的可画物，前后缓存可以通过调用 glXSwapBuffers()命令进行交换。该命令中包含了一个隐含的 glFlush()命令。

附 4.2.5 使用 X 字体

函数 glXUseXFont()提供了在 OpenGL 中使用 X 字体的捷径。

附 4.3 GLX 库函数原型

初始化：

Bool glXQueryExtension (Display *dpy, int *errorBase, int *eventBase);
Bool glXQueryVersion (Display *dpy, int *major, int *minor);
XVisualInfo* glXChooseVisual (Display *dpy, int screen, int *attribList);
int glXGetConfig (Display *dpy, XVisualInfo *vis, int attrib, int *value);

控制渲染：

GLXContext glXCreateContext (Display *dpy, XVisualInfo *vis, GLXContext shareList, Bool direct);
void glXDestroyContext (Display *dpy, GLXContext ctx);
void glXCopyContext (Display *dpy, GLXContext src, GLXContext dst, GLuint mask);

Bool glXIsDirect (Display *dpy, GLXContext ctx);
Bool glXMakeCurrent (Display *dpy, GLXDrawable draw, GLXContext ctx);
GLXContext glXGetCurrentContext (void);
GLXDrawable glXGetCurrentDrawable (void);
GLXPixmap glXCreateGLXPixmap (Display *dpy, XVisualInfo *vis, Pixmap pixmap);
void glXDestroyGLXPixmap (Display *dpy, GLXPixmap pix);
void glXWaitGL (void);
void glXWaitX (void);
void glXSwapBuffers (Display *dpy, Window window);

使用 X 字体：

---

void glXUseXFont (Font font, int first, int count, int listBase);

附录 5　OpenGL 辅助库

本附录简要地描述了 OpenGL 辅助库（GLAUX）。它有以下几个主要部分：

- 初始化和退出窗口。
- 处理窗口和输入事件。
- 装载颜色图。
- 初始化及绘制三维物体。
- 管理后台进程。
- 运行程序。

借助辅助库应用程序就可以构造事件处理机制，以便使用回调函数。例如，首先打开一个窗口，为特定事件注册一个回调函数，然后创建程序的主循环。在主循环中，如果事件发生，为其注册的回调函数就被执行。在回调函数完成时，控制流返回主循环。

附 5.1　初始化及退出窗口

初始化过程中，一般需要用到的函数有下面几个：

---

void auxInitWindow(GLbyte *titleString);

利用 auxInitDisplayMode()和 auxInitPosition()命令打开一个指定的特殊窗口。

---

void auxInitDisplayMode(GLbitfield mask);

通知 auxInitWindow()命令是否创建一个 RGBA 窗口或颜色索引窗口，以及是单缓存还是多缓存窗口。

---

void auxInitPosition(GLint x, GLint y, GLsizei width, GLsizei height);

通知 auxInitWindow()命令在屏幕的什么地方设置一个窗口。

附 5.2 处理窗口和输入事件

下面几个函数用于为特定的时间注册回调函数。

void auxReshapeFunc(void (*function)(GLsizei, GLsizei));

为窗口尺寸改变、窗口移动及窗口暴露事件注册回调函数。

void auxKeyFunc(GLint key, void (*function)(void)) ;

为键盘事件注册回调函数。

void auxMouseFunc(GLint button, GLint mode, void (*function)(AUX_EVENTREC *));

为鼠标事件注册回调函数。

附 5.3 装载颜色图

GLAUX 提供了通用的函数 auxSetOneColor()，它能以 RGB 值来加载颜色索引。

void auxSetOneColor(GLint index, GLfloat red, GLfloat green, GLfloat blue);

在颜色图中以给定的红色、绿色和蓝色值加载颜色索引 index。

附 5.4 初始化和绘制三维物体

下面是一些由辅助库提供的绘图子程序。每一组有两个函数，一个绘制实体，一个绘制线框图。

绘制球体：

void auxWireSphere(GLdouble radius);
void auxSolidSphere(GLdouble radius);

绘制立方体：

---

void auxWireCube(GLdouble size);
void auxSolidCube(GLdouble size);

绘制方盒：

---

void auxWireBox(GLdouble width, GLdouble height, GLdouble depth);
void auxSolidBox(GLdouble width, GLdouble height, GLdouble depth);

绘制圆环：

---

void auxWireTorus(GLdouble innerRadius, GLdouble outerRadius);
void auxSolidTorus(GLdouble innerRadius, GLdouble outerRadius);

绘制圆柱：

---

void auxWireCylinder(GLdouble radius, GLdouble height);
void auxSolidCylinder(GLdouble radius, GLdouble height);

绘制二十面体：

---

void auxWireIcosahedron(GLdouble radius);
void auxSolidIcosahedron(GLdouble radius);

绘制八面体：

---

void auxWireOctahedron(GLdouble radius);
void auxSolidOctahedron(GLdouble radius);

绘制四面体：

---

void auxWireTetrahedron(GLdouble radius);
void auxSolidTetrahedron(GLdouble radius);

绘制十二面体：

void auxWireDodecahedron(GLdouble radius);
void auxSolidDodecahedron(GLdouble radius);

绘制圆锥：

void auxWireCone(GLdouble radius, GLdouble height);
void auxSolidCone(GLdouble radius, GLdouble height);

绘制茶壶：

void auxWireTeapot(GLdouble size);
void auxSolidTeapot(GLdouble size);

附 5.5 管理后台进程

可以定义一个函数，在没有其他待处理的事件时，则执行该函数。该函数有惟一的参数，是一个指向该函数的指针。

void auxIdleFunc(void *func);

定义函数 func，在没有其他待处理的事件时，执行该函数。

附 5.6 运行程序

函数 auxMainLoop()用于指定绘制场景子函数。

void auxMainLoop(void(*displayFunc)(void));

指定函数 displayFunc，当窗口更新时调用该函数。

附录 6　计算法线向量

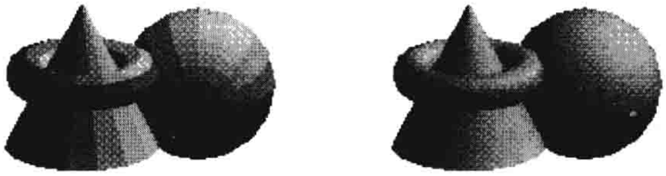
本附录描述了如何计算表面法线向量。要使用 OpenGL 的光照工具，就必须定义法线。本附录详细地介绍法线向量的计算。本附录的主要内容如下：

- 为解析表面确定法线。
- 由多边形数据确定法线。

由于法线垂直于表面，因此可以在表面上特定点处，通过先找到与该点表面相切的平面来确定法线，法线是垂直该平面的一个向量。例如，对于理想的球体，表面法线与球心指向该点的向量方向相同。对于其他类型的表面，有更好的方法来确定法线，当然这是依赖于表面是如何定义的。

我们知道，平滑曲面是由大量的小的平面多边形来近似的。在这种近似中，垂直于各多边形的向量就认为是表面的法线。当然，在许多情况下对曲面都有较确切的数学描述，真实表面的法线在每一点都可以计算出来。使用真实法线明显地改善了渲染质量，如附图 6-1 所示。本附录的两个主要部分分别描述了在两种情况下如何计算法线向量。

- "为解析表面确定法线"，解释了如何确定用数学方法描述的表面的法线。
- "由多边形数据确定法线"，概括了用多边形数据描述表面时如何确定法线。



附图 6-1　用真实法线和多边形法线进行渲染的对比

附 6.1　为解析表面确定法线

解析表面是指平滑可微的，可用数学方程或方程组来描述的表面。在许多情况下，确定法线的最简单的表面就是解析表面，对这种表面可用下式显式定义：

$V(s,t) = [\ X(s,t)\quad Y(s,t)\quad Z(s,t)\ ]$

其中，s 和 t 限制在某区域内，X，Y，Z 是具有两个自变量的可微函数。为了计算这个表面法线，需要确定：

$$\frac{\partial V}{\partial s} \text{及} \frac{\partial V}{\partial t}$$

分别为该表面在 s 和 t 方向的切向量。其叉积：

$$\frac{\partial V}{\partial s}\times\frac{\partial V}{\partial t}$$

与两个向量垂直，因此它也垂直于表面。下式说明了如何计算两个切向量的叉积。

$$[V_x\,V_y\,V_z]\times[W_x\,W_y\,W_z]=[\,V_yW_z-V_zW_x \quad V_zW_x-V_xW_z \quad V_xW_y-V_yW_x]$$

结果要进行归一化。为了对向量[x,y,z]进行归一化，首先计算其长度：

$$\text{length}=\sqrt{x^2+y^2+z^2}$$

然后用长度去除每一个分量。

下面看一个例子。考虑解析表面：

$$V(s,t) = [\ s^2 \quad t^3 \quad 3\text{-}st\ ]$$

根据表面方程，有：

$$\frac{\partial V}{\partial s}=[2s \quad 0 \quad -t],\frac{\partial V}{\partial t}=[0 \quad 3t^2 \quad -s],\frac{\partial V}{\partial s}\times\frac{\partial V}{\partial t}=[-3t^3 \quad 2s^2 \quad 6st^2]$$

所以，比如当 s=1 且 t=2 时，表面上相应的点为(1, 8, 1)，而且向量(-24, 2, 24)在该点处垂直于表面。这一向量的长度为 34，所以单位法线向量为(-24/34, 2/34, 24/34) = (-0.70588, 0.058823, 0.70588)。

对于用隐函数定义的解析表面，如 $F(x, y, z) = 0$，问题就变得困难了。在某些情况下，仍可以计算出其中一个变量，比如，$z = G(x, y)$，然后将它转换为前面给出的显式的形式：

$$V(s,t)=[s \quad t \quad G(s,t)]$$

然后如前所述接着进行计算。

如果无法得到显式的表面方程，可以利用这样的事实，即用梯度表示法线向量：

$$\nabla F=[\frac{\partial F}{\partial x} \quad \frac{\partial F}{\partial y} \quad \frac{\partial F}{\partial z}]$$

在特定点(x, y, z)处求值。计算梯度可能是简单的，但是确定表面上的点则比较困难。我们看一个隐式表示的解析表面的例子。考虑中心在原点处，半径为 1 的球体方程：

$$x^2 + y^2 + z^2 - 1 = 0$$

将方程变为：

$$F(x, y, z) = x^2 + y^2 + z^2 - 1$$

从上式可以解出 z：

$$z=\pm\sqrt{1-x^2-y^2}$$

因此法线可以显式地计算：

$$V(s,t)=[s \quad t \quad \sqrt{1-s^2-t^2}]$$

步骤如前所述。

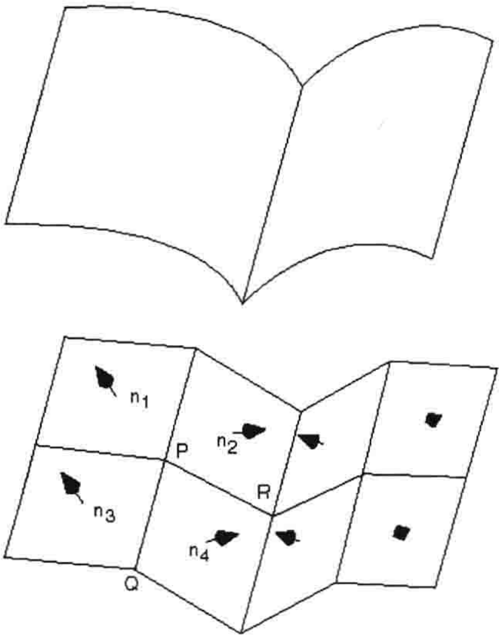
如果不能解出 z，你可能需要用到梯度

$\nabla F = [2x \quad 2y \quad 2z]$

前提是可以在表面上找到一点。在上例中，找到一个点并不是特别困难，例如，(2/3, 1/3, 2/3)就在表面上。根据梯度可以计算出这一点处的法线就是(4/3, 2/3, 4/3)，单位法线是(2/3, 1/3, 2/3)，与在表面上的点相同。

附 6.2 由多边形数据确定法线

如前所述，通常要为由多边形数据描述的表面确定法线，使多边形看起来更平滑。在多数情况下，最简单的办法就是计算出每一个多边形面的法线向量，然后与相邻表面的法线取平均值。在相邻表面的共同顶点处使用平均法线。附图 6-2 显示了一个表面及其多边形近似。当然，如果这个多边形代表了确切的表面而不是一种近似，如一个立方体或一个切割的钻石，那么就用不着进行平均。表面的法线按下面描述的方法进行计算，然后同一个表面的各顶点使用相同的法线。



附图 6-2 对法线取平均

要确定平面多边形的法线，取多边形上任意的不在一条直线上的三个点 v1，v2 和 v3。叉积[v1 - v2] × [v2 - v3]垂直于此多边形。（通常要将叉积结果向量归一化）。在将相邻表面法线向量归一化时，要避免给其中的一个向量过大的权值。例如，在附图 6-2 中，如果 n1，n2，n3 和 n4 分别是相交于点 P 处的四个多边形的法线，计算出 n1+n2+n3+n4 然后对其进行归一化。结果向量可以用作点 P 的法线向量。

有时，在特殊情况下对这种方法要稍加改变。例如，在表面的边界处（如附图 6-2 中的点 Q），需要基于表面的形状选择一个更好的法线。有时在边界处也只是将多边形法线进行平均。类似的，某些模型既有平滑部分又有尖角部分（如附图 6-2 中的点 R）。在这种情况下，折缝两侧的法线不要取平均。相反，在折缝一侧的多边形用一个法线绘制，而另一侧的多边形则用另一条法线来绘制。

附录 7　齐次坐标与变换矩阵

本附录部分将详细讨论齐次坐标，同时也列出用于旋转、放大、平移、透视投影和正交投影的变换矩阵形式。这些主题曾在第 4 章中介绍并讨论过。尽管投影几何的内容涉及各个维数，但在下面的论述中，齐次坐标特指三维齐次坐标。

本附录由以下主要部分组成：

- 齐次坐标。
- 变换矩阵。

附 7.1　齐次坐标

OpenGL 函数通常是针对二维或三维物体的顶点的，但实际上，在 OpenGL 内部，将所有顶点都看作是有四个坐标值的三维齐次顶点。每一列的向量$(x, y, z, w)^T$中，如果至少有一个元素不为零，此列向量就表示一个齐次顶点。如果实数 a 不为零，则$(x, y, z, w)^T$与$(ax, ay, az, aw)^T$表示同一个顶点（类似于：$x/y = (ax)/(ay)$）。一个三维的欧几里得空间点$(x, y, z)^T$变成坐标为$(x, y, z, 1.0)^T$的齐次顶点，二维的欧几里得空间点$(x, y)^T$变成坐标为$(x, y, 0.0, 1.0)^T$的齐次顶点（T 表示向量的转置）。

只要 w 不为零，齐次顶点$(x, y, z, w)^T$ 就对应于三维空间点$(x/w, y/w, z/w)^T$ 。$w = 0.0$ 则对应于非欧几里得空间点，也可以认为它是某种理想化的“无穷远点”。为了理解这种无穷远点的概念，考虑以下各点(1, 2, 0, 1)，(1, 2, 0, 0.01)和(1, 2.0, 0.0, 0.0001)，它们对应于欧几里得空间点(1, 2), (100, 200)与(10000, 20000)，这些点沿着直线 $2x = y$ 趋向于无穷大。因此，可以认为(1, 2, 0, 0)是沿直线方向到达的无穷远点。

注意：OpenGL 无法正确处理 $w < 0$ 时的齐次裁剪坐标。为了确保程序代码可以用于各种 OpenGL 系统，w 一定要是非负值。

附 7.2　变换

附 7.2.1　顶点变换

顶点变换（如旋转、平移、比例变换和剪切）与投影（如透视投影和正交投影），都可以通过对代表顶点的坐标应用一合适的 4×4 矩阵来表示。如果 v 表示一个齐次顶点，M 是一个 4×4 变换矩阵，则 Mv 是经 M 变换的 v 图像。在计算机图形应用软件中，所用的变

换通常是非奇异的，也就是说矩阵 M 可逆。这并不是必需的，但使用奇异矩阵会带来一些问题。

经过变换之后所有转换的顶点都进行剪裁，使 x, y 和 z 在[-w, w]的范围内（假定 $w > 0$）。注意这个范围对应于欧几里得空间的[-1.0, 1.0]。

附 7.2.2 法线变换

法线向量的变换与顶点或位置向量的变换不同。从数学角度讲，最好不把法线作为向量，而是作为垂直于向量的平面。因此，对法线向量的变换规则将表示为垂直平面的变换。

一个齐次平面由行向量(a , b, c, d)表示，其中 a, b, c, d 中至少有一项不为零。如果 q 是非零实数，则(a, b, c, d)与(qa, qb, qc, qd)表示同一平面。如果 $ax+by+cz+dw = 0$，说明点$(x, y, z, w)^T$在平面(a, b, c, d)上。如果 $w = 1$，这就表示一个标准的欧几里得平面。为了使(a, b, c, d)能表示一个欧几里得平面，a, b, c 中至少要有一项不为零。如果它们全为零，(0, 0, 0, d)就表示“无数个点”组成的“无限的平面”。

如果 p 是齐次平面而 v 是齐次顶点，那么命题“v 在 p 上”在数学上表示为：$pv = 0$，这里 pv 是法线矩阵相乘。如果 M 是一个非奇异的顶点变换（即 M 是一个可逆的 4×4 矩阵），则 $pv = 0$ 等价于 $pM^{-1}Mv = 0$，所以 Mv 取决于平面 pM^{-1}。因此，pM^{-1}是顶点变换 M 下的平面图像。

如果习惯于将法线作为向量，而不是当作平面的垂线，则取 v 和 n 两个向量，v 垂直于 n。这样 $n^T v = 0$。因此，对一个任意的非奇异变换 M 有 $n^T M^{-1}Mv = 0$，表示 $n^T M^{-1}$是法线向量变换的转置矩阵。这样，变换后的法线向量是$(M^{-1})^T n$。换一句话说，法线向量的变换用变换点的变换矩阵的逆阵的转置实现。

附 7.2.3 变换矩阵

尽管任何非奇异的矩阵 M 都表示一个有效的投影变换，但几个特殊的矩阵更为常用。这些矩阵在下面中给出。

1. 平移

调用函数 glTranslate*(x, y, z)产生 T，其中：

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. 比例变换

调用函数 glScale*(x, y, z)产生 S，其中：

$$S=\begin{bmatrix}x&0&0&0\\0&y&0&0\\0&0&z&0\\0&0&0&1\end{bmatrix}\qquad S^{-1}=\begin{bmatrix}\frac{1}{x}&0&0&0\\0&\frac{1}{y}&0&0\\0&0&\frac{1}{z}&0\\0&0&0&1\end{bmatrix}$$

注意，只有 x, y, z 都不为零时，S^{-1} 才存在。

3. *旋转*

调用函数 glRotate*(a, x, y, z) 产生 R 如下：

令 $v = (x, y, z)^T$, $u = v/\|v\| = (x', y', z')$。

并令：

$$S=\begin{bmatrix}0&-z'&y'\\z'&0&-x'\\-y'&x'&0\end{bmatrix},\qquad M=uu^T+(\mathrm{Cos}\,\alpha)(1-uu^T)+(\mathrm{Sin}\,\alpha)S$$

那么：

$$R=\begin{bmatrix}m&m&m&0\\m&m&m&0\\m&m&m&0\\0&0&0&1\end{bmatrix}$$

其中，m 表示 M 中的元素，为 3 阶方阵。

矩阵 R 总是有定义的。如果 x=y=z=1，那么 R 是单位矩阵。可以用-a 代替 a 或做变换得到 R 的逆阵 R^{-1}。函数 glRotate*()产生一个绕任意轴旋转的矩阵。通常都是相对某一坐标轴旋转，相应的矩阵如下：

$$\text{GlRotate*(2, 1, 0, 0):}\quad\begin{bmatrix}1&0&0&0\\0&\cos\alpha&-\sin\alpha&0\\0&\sin\alpha&-\cos\alpha&0\\0&0&0&0\end{bmatrix}$$

$$\text{GlRotate*(2, 0, 1, 0):}\quad\begin{bmatrix}\cos\alpha&0&\sin\alpha&0\\0&1&0&0\\-\sin\alpha&0&\cos\alpha&0\\0&0&0&1\end{bmatrix}$$

$$\text{GlRotate*(2, 0, 0, 1):}\begin{bmatrix}\cos\alpha & -\sin\alpha & 0 & 0\\ \sin\alpha & \cos\alpha & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

4. 透视投影

调用函数 glFrustum(l, r, b, t, n, f)产生 R，其中：

$$R=\begin{bmatrix}\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0\\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0\\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n}\\ 0 & 0 & -1 & 0\end{bmatrix}$$

$$R^{-1}=\begin{bmatrix}\frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n}\\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n}\\ 0 & 0 & 0 & \frac{n+n}{2n}\\ 0 & 0 & \frac{-(f+n)}{f-n} & 0\end{bmatrix}$$

只要 l 不等于 r, t 不等于 b, n 不等于 f，R 就是有定义的。

5. 正交投影

调用函数 glOrtho*(l, r, b, t, u, f)产生 *R*，其中：

$$R=\begin{bmatrix}\frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l}\\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b}\\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n}\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$R^{-1}=\begin{bmatrix}\frac{r-l}{2} & 0 & 0 & \frac{r+l}{2}\\ 0 & \frac{t-b}{2} & 0 & \frac{t+b}{2}\\ 0 & 0 & \frac{f-n}{2} & \frac{n+f}{2}\\ 0 & 0 & 0 & 1\end{bmatrix}$$

只要 l 不等于 r, t 不等于 b, n 不等于 f，R 就是有定义的。

附录 8　OpenGL 编程要点

本附录给出了一些对编程有用的指导性技巧。但要记住，这些技巧仅仅是基于 OpenGL 设计者的意图，并不是实际应用的经验，也并没有实现。本附录包括如下主要内容：

- OpenGL 正确编程技巧。
- 提高 OpenGL 程序性能。
- GLX 技巧。

附 8.1　OpenGL 正确编程技巧

- 不要依赖于 OpenGL 实现的错误行为。因为在将来的 OpenGL 版本中有可能会改变。例如，OpenGL 1.0 忽略在 glBegin() 和 glEnd()之间的矩阵操作调用，而 OpenGL 1.1 则不然。换一种说法，OpenGL 的错误语义会以向上兼容的形式随版本变化。
- 要用投影矩阵将所有几何图形投影到一个平面上。如果使用模型视图矩阵，在人眼坐标系中操作的 OpenGL 特征（例如光照和应用程序所定义的裁剪平面）将会失效。
- 不要频繁地改变某些矩阵。例如在实现旋转时，最好不要以一个不断增加的角度反复调用 glRotate()函数，正确的方法是用 glLoadIdentity()函数初始化每帧画面给定的矩阵，然后用给定的角度调用 glRotate()函数。
- 在定义显示序列表时不要期望得到出错信息。只有当列表被执行时，显示列表中的命令才会产生出错信息。
- 将较近的视景台体放在离视点尽可能远的地方，这样可以优化深度缓存操作。
- 调用 glFlush()函数可以强制先前的 OpenGL 命令被执行。不要指望 glGet*()或 glIs*()函数来更新渲染流。
- 在渲染预抖动图像时将抖动操作关闭（例如，在调用 glCopyPixels()函数时要将抖动操作关闭）。
- 有效利用累积缓存。
- 如果要实现精确的二维光栅效果，必须仔细地设置正交投影和要被光栅化的原语顶点。正交投影可以像下例这样用整数坐标指定：

  gluOrtho2D(0，width，0，height);

  其中，width 和 height 是视区尺度。假定投影矩阵、多边形顶点以及像素图像位置应该以整数坐标放置以便预先光栅化。例如，glRecti(0, 0, 1, 1)填充视区左下角的像素，glRasterPos2i(0, 0)将未缩放的图像定位在视区左下角。

允许所有原语由整数位置设定，同时仍能确保预先光栅化的一个最适宜的折衷方案就是将 x 和 y 平移 0.375，如下例：

```
glViewport(0, 0, width, height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0, width, 0, height);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.375, 0.375, 0.0);
/* render all primitives at integer positions */
```

- 避免使用负的 w 顶点坐标和负的 q 纹理坐标。如果用这样的坐标描述原语，OpenGL 就无法正确地剪取坐标，而产生解释错误。

附 8.2　提高 OpenGL 程序性能

- 为了提高速度，在只有一个材质属性改变时应该使用 glColorMaterial()函数。而在不经常变化，或多于一个材质特征要被改变时使用 glMaterial()函数。
- 用 glLoadIdentity()函数初始化一个矩阵，而不要加载单位矩阵的拷贝。
- 使用指定的矩阵调用，如 glRotate*()，glTranslate*()和 glScale*()等，而不要调用 glMultMatrix()函数来组织自己的旋转、平移和比例变换矩阵。
- 用 glPushAttrib()和 glPopAttrib()存储和恢复状态值。只有应用程序中需要使用状态值计算时才调用查询函数。
- 使用显示列表来减小状态改变引起的开销。
- 使用显示列表来封装需要重复绘制的刚体的渲染调用。
- 即使是简单的表面分化也应尽量使用求值程序，以减小客户/服务器环境下的网络带宽。
- 尽量使用单位法线，避免过分使用 GL_NORMALIZE。在光照计算时避免使用 glScale*()函数，因为该函数几乎总要激活 GL_NORMALIZE。
- 如果不要求平滑的明暗效果，将 glShadeModel() 设定为 GL_FLAT。
- 可能的情况下每帧调用一次 glClear()函数。不要使用 glClear()函数来清除缓存中的小的子区域；仅在全部清除时才使用此函数。
- 调用 glBegin(GL_TRIANGLES)一次性绘制独立的多个三角形，而不要多次调用 glBegin(GL_TRIANGLES)，或调用 glBegin(GL_POLYGON)函数。即使仅绘制一个三角形时也要用 GL_TRIANGLES 参数值，而不是 GL_POLYGON 参数值。在同样方式下应尽量使用 glBegin(GL_QUADS)函数，而不是反复调用 glBegin(GL_POLYGON)

函数。同样地，用一个 glBegin(GL_LINES)函数调用绘制多个独立线段，而不要多次调用 glBegin(GL_LINES)函数。

- 一般地，用向量形式的命令传递预先计算的数据，而在使用这些数据时用标量形式的命令。
- 避免进行多余的模式改变，例如给一个多边形的各顶点重复设置同一颜色值。
- 在绘制或拷贝图像时，要确保禁止代价昂贵的光栅化及片元操作。

附 8.3 GLX 的技巧

- 用 glXWaitGL()函数来强制使 X 渲染指令在 GL 渲染指令之后完成，而不要使用 glFinish()函数。
- 同样地，用 glXWaitX()函数，强制 GL 渲染指令跟随在 X 渲染指令之后完成，而不要使用 glXSync()函数。
- 在使用 glXChooseVisual()函数时要小心，因为二值选择需要准确匹配。

附录 9　OpenGL 的不变性

OpenGL 没有精确的像素规范，因此，它不能保证不同的 OpenGL 实现产生的图像之间的匹配。然而，在某些情况下，OpenGL 确实指明了同一实现所生成的图像的精确匹配。本附录描述了对这些情况进行定义的不变性规则。

最明显最基本的情况就是重复性，在同样的初始条件下发出一组命令时，一致的 OpenGL 实现将产生相同的结果。尽管重复性对检测和验证是很有用的，但对程序员并没有用处，因为安排相同的初始条件是非常困难的，比如：将一个视景渲染两次，在交换前后缓存之后，第二次渲染结果根本不能满足要求。因此重复性不能用来保证产生稳定的具有双缓冲的图形。

依靠不变性来执行的一个简单易行的算法就是以背景颜色重画直线，将直线擦除。OpenGL 要求光栅化产生的片原坐标相对于帧缓存内容、矩阵值、剪切参数、写操作掩码、所有的清除值、当前颜色、索引、法线、纹理坐标、边缘标记值、当前光栅颜色、光栅索引、光栅纹理坐标、材料属性等值是不变的。不变性规则还要求确切地生成相同的片元，包括片元颜色值、剪切参数、写操作掩码等。

OpenGL 建议，生成的片元相对于矩阵模式、矩阵堆栈深度、alpha 测试参数、模板参数、深度测试参数、融合参数、逻辑操作、像素存储变换参数是不变的。

最后，OpenGL 要求片元算法，如混合及深度测试算法，相对于 OpenGL 的状态变量具有不变性。

所有不变性要求的结果，使得 OpenGL 能够保证利用相同的命令序列同时或分别渲染到不同的颜色缓存中的图像具有相同的像素值。

附录 10　术语表

A

走样（aliasing）

指一种渲染技术，把需要进行渲染的图元的颜色赋值给像素点，不论图元是覆盖了整个像素区域还是仅仅覆盖了一部分。走样的结果将产生锯齿状的边缘，或锯齿状图形。

alpha

第四种颜色分量。alpha 颜色分量并不直接显示。其典型的应用是控制颜色的混和。习惯上，OpenGL 的 alpha 分量对应着不透明的程度，也就是说 alpha 值为 1.0 表示完全不透明，而 0.0 表示完全透明。

动画（animation）

该过程是指对视景不断进行渲染，以足够快的速度平稳地改变视点或（和）物体位置来实现运动的幻觉。OpenGL 的动画几乎全部是通过双缓冲器来实现的。

反走样（antialiasing）

指一种渲染技术，根据需要渲染的图元所覆盖的像素区域大小来分配像素颜色。反模糊化渲染减少或消除了模糊化渲染产生的锯齿图形。

特定应用裁剪（application-specific clipping）

指在人眼坐标中对相对于某平面的图元裁剪。该平面由 glClipPlane( )函数定义的应用程序指定。

B

背面（back faces）

参见面（face）。

位（bit）

二进制数。一个声明的位变量只有两个可能值：0 或 1。二进制数字是由一个或多个位（比特）构成的。

位图（bitmap）

位图，即位的矩形阵列。同样，由 glBitmap( )命令渲染的图元也是位图，该命令将位图参数作为掩码使用。

位平面（bitplane）

与像素一一对应所组成的矩形阵列。帧缓冲器由多个位平面组成。

混合（blending）

将两种颜色分量混成变为一种颜色分量，通常是在这两个颜色分量之间做线性插值。

缓存（buffer）

又称缓冲器，用于存储单一变量（如深度或绿色）或单一索引（如颜色索引或模板索引）的一组位平面。在一些情况下，红色、绿色、蓝色和 alpha 各单缓冲器统一为一个颜色缓冲器。

C

客户（client）

指发送 OpenGL 命令的计算机。发出 OpenGL 命令的计算机可以通过网络连接，在另一台计算机上执行命令，或者也可在本地计算机上执行。参见服务器（Server）。

客户内存（client memory）

客户计算机的主内存（程序变量存储其中）。

裁剪坐标（clip coordinates）

在透视分割之前，投影矩阵变换之后的坐标系。视图体的剪辑是在裁剪坐标系中进行的，但特定应用的裁剪并非如此。

裁剪（clipping）

指除去几何图元中裁剪平面定义的半空间之外的部分的过程。对于点，则简单地擦除定义空间外的点；对于一条直线或多边形，除擦除定义空间外的部分之外，还要附加生成必要的顶点，以保持裁剪的半空间内图元的完整性。几何图元和当前的光栅位置（当有定义时）常常按照由左部、右部、底部、顶部、近处和远处的视图体的平面来裁剪。应用程序中，可以指定可选择的、应用于人眼坐标的特定应用裁剪平面。

颜色索引（color index）

指一个单一的数值，代表颜色的名字而不是颜色的值。OpenGL 的颜色索引在操作中（如插补和抖动在颜色索引状态下运行时）可以被看作连续的值（例如，浮点型值）。当然，存储在帧缓冲器中的颜色索引常常是整型值。浮点型索引经四舍五入转换成整型值。

颜色索引模式（color-index mode）

如果颜色缓冲区储存颜色索引的话，OpenGL 的内容表则处在颜色索引模式下，而不是红色、绿色、蓝色和 alpha 颜色成分。

颜色图（color map）

指索引到 RGB 的映射表，通过显示硬件进行访问。每一个颜色索引从颜色缓存中读取，通过在颜色图中查找转化成一个 RGB 的三元组，并发送到监视器。

命令（command）

OpenGL 库的基本组成单位。

分量（components）

代表强度或数量的单一的、连续的（如浮点型）数值。尽管有时用到其他的标准，通常，一个 0 分量代表最小的数值或强度，而分量数值为 1 代表强度或数值的最大值。由于分量值是在归一化的范围内解释的，因此它的确定是和实际的分辨率无关的。例如，无论颜色缓冲区是否分别存储了 4 位、8 位、12 位，RGB 三元组（1，1，1）都表示白色。

域外分量常常夹在归一化范围内，而不是去掉或说明。例如 RGB 三元组（1.4, 1.5, 0.9）在更新颜色缓冲区前归一为（1.0, 1.0, 0.9）。红色、绿色、蓝色、alpha 和深度常被看成是分量而不是索引。

内容表（context）

指一个完整的 OpenGL 状态变量系列。注意帧缓冲区的内容并不是 OpenGL 的一个部分，但是其配置是 OpenGL 的一部分。

凸多边形（convex）

如果多边形平面内的任何直线与多边形相交都不多于两次，那么这个多边形就是凸的。

凸壳体（convex hull）

最小的凸区域包括一族指定的点。在二维情况下，凸壳体在概念上是这样建立的：环绕某些点伸展橡胶带，使这些点都处在橡胶带以内。

消隐（culling）

一个清除多面体的前表面或后表面使其不被显示的过程。

当前矩阵（current matrix）

指一个将坐标从一坐标系转换到另一坐标系的矩阵。在 OpenGL 中有三个转移矩阵：模型（式）视图矩阵由物体坐标（由用户定义的坐标）转换到人眼坐标；透视矩阵由人眼坐标转换到裁剪坐标，纹理矩阵由特定或发生纹理坐标转换成矩阵描述。这三个（堆栈）转移矩阵都可以通过 OpenGL 矩阵操作命令来进行操作。

当前光栅位置（current raster position）

是一个窗口坐标位置，在光栅化时确定出图像图元的放置。当前的光栅位置和其他的当前的光栅参数，在调用 glRasterPos( )时更新。

D

深度（depth）

通常指 Z 窗口坐标。

深度提示（depth-cuing）

指一种渲染技术，根据离视点的距离对颜色进行赋值。

显示列表（display list）

一个有命名的 OpenGL 命令列表。显示列表常存储在服务器里，因此显示列表可以用于减缓客户/服务器网络环境的繁忙程度。显示列表的内容可以进行预处理，因此相同的一组 OpenGL 命令在显示列表中比在直接模式中执行更加有效。这种预处理对于高级命令的计算，如 glTexImage( )，显得特别重要。

抖动（dithering）

指一种技术，用来在牺牲空间分辨率的代价下增加在一幅图像中可感知的颜色范围。对相邻的像素点赋以不同的颜色值，当以一定的距离来观看时，这些颜色看起来好像是混合而成的一种颜色。这一技术类似于在黑白印刷物里通过半色调来获得灰色的阴影。

双缓冲区（double-buffering）

同时带有正面颜色缓存和背面颜色缓存的 OpenGL 内容表称为双缓冲区。平滑的动画是通过只渲染背面缓存（但并不显示）来实现的，然后交换正背缓存。

E

要素（element）

一个分量或索引。

定值（evaluated）

在 OpenGL 中，由预先指定的贝塞尔方程产生物体坐标顶点和参数的过程。

人眼坐标（eye coordinates）

在投影矩阵变换之前，模型视图矩阵转移之后的一种坐标系。光照和特定应用的裁剪在人眼坐标中完成。

F

面（face）

多边形的一侧。每一个多边形有两个“面”：正面和背面。在窗口中只有一个面是可见的。当此多边形投影到窗口中后，就可以有效地确定是正面可见还是背面可见。如果多边形的边是顺时针的，则一个面可见；如果是逆时针，则另一面可见。究竟是正面还是背面可见，由 OpenGL 程序员决定。

无明暗效果的阴影（flat-shaded）

指对整个区域的一个图元着色，采用单一恒定的一种颜色，而不是平稳的插值颜色。参见 Gouraud 阴影。

雾（fog）

指一种基于观看者的距离，将目标颜色褪色为背景颜色渲染技术，可以用来模拟大气效应如薄雾、尘雾、烟雾。给定深度提示，雾常常可以用来辅助实现观察者的距离感觉。

字体（fonts）

一组图形化的字符表示，常用于显示文本字符串。这些字符可以是罗马字母、数学符号、亚洲表意字符、埃及象形文字等等。

片原（fragment）

片原是由图元的光栅化产生的。每一个片原对应于一个单像素，包括颜色、深度，有时包括纹理坐标值。

帧缓存（framebuffer）

一个给定窗口或内容表的所有缓冲器。有时包括图形硬件加速器的所有像素内存。

正面（front faces）

参见面（face）。

平截台体（frustum）

经透视分割扭曲了的视图体。

G

GAMMA 修正（gamma correction）

指用于存储在帧缓冲器内的颜色的一个函数，用来把眼睛（有时是显示器）的非线性响应修正为颜色深度值的线性变化。

几何模型（geometric model）

指描述对象的物体坐标顶点和参数。注意 OpenGL 并不是为几何模型定义一个语法，而是为渲染几何模型定义语法和语义。

几何物体（geometric object ）

即几何模型。

几何图元（geometric primitive）

指一个点、一条线或是一个多边形。

Gouraud 阴影（Gouraud shading）

指在一个多边形或是线段内的平稳的颜色内插。在顶点对颜色赋值，并沿图元线性插值，产生相对平滑的颜色变化。也称为平滑阴影。

族（group）

在客户内存中由一族（一个、两个、三个或四个）要素来表达的一幅图像中的每一个像素。因此，在每一个客户内存图像的内容表中，一族或一个像素是一回事。

H

半空间（half-spaces）

一个平面将空间分为两个半空间。

齐次坐标（homogeneous coordinates）

在 n 维投影空间中用来表示点的 n+1 个坐标的集合。投影空间中的点可以被看作是欧几里德空间内的点，同时认为某些点在无限远处。这一坐标系是齐次的，因为每一个坐标以一个相同的非零常量进行比例变换，这一变换并不随坐标所指向的点的变化而变化。齐次坐标在投影几何学的计算中非常有用，在需要把视景投影到一个窗口上的计算机图形学中也显得非常重要。

I

图像（image）

指像素的矩形阵列，存在于用户内存中或帧缓冲器中。

图像图元（image primitive）

指一幅位图或一幅图像。

直接模式（immediate-mode）

在命令被直接调用时执行它，而不是在显示列表中执行。并不存在直接模式位，直接模式的模式指的是 OpenGL 的应用，而不是一个 OpenGL 状态的特定位。

索引（index）

一个被理解为绝对值的单值，而不是在特定范围内的一个归一化值（类似一个分量）。颜色索引是指颜色的名字，使用颜色图时，显示硬件并不引用颜色名。当索引超出范围时，通常被认为是无效的而被屏蔽掉。例如，索引 0xf7 在写入一个 4 位的缓存（颜色或模板）时，被屏蔽为 0x7。颜色索引和模板索引都看成是索引而不是分量。

IRIS GL

Silicon Graphics 公司的专利图形库，从 1982 年到 1992 年间发展起来。OpenGL 是以 IRIS GL 为起点设计的。

锯齿（jaggies）

模糊化渲染的前身。用模糊化渲染的图元的边缘是锯齿状而不是平滑的。例如：一条接近水平的模糊化直线，是作为一系列的水平线来渲染相邻的像素列，而不是看成一条平滑、连续的线。

L

光照（lighting）

基于当前光、材料属性和光照模型模式来计算顶点的颜色的过程。

线（line）

指两个顶点间的有限宽度的整齐区域（不像数学上的线，OpenGL 的线有有限的宽度和长度）。线上的每一个组元本身是一条线。

亮度（luminance）

指表面的可以感觉到的光亮程度。常常是一个由红色，绿色和蓝色值的加权平均数值给出的综合的感觉亮度。

M

矩阵（matrix）

指一个二维的数组。OpenGL 矩阵都为 4 阶方阵，尽管存储在用户内存中时他们被看成是 1*16 的一维数组。

模型视图矩阵（modelview matrix ）

指一 4 阶方阵，将对象坐标下的点、线、多边形和光栅位置转换成人眼坐标。

显示器（monitor）

指显示帧缓冲器里的映像的设备。

运动模糊化（motion blurring）

是指一种技术，在拍摄运动物体时，或是以运动的照相机拍摄一个静止物体时，利用这种技术模拟电影效果。在缺少模糊化运动的动画中，运动物体将表现出跳变闪烁。

N

非凸多边形（nonconvex）

如果多边形的平面内存在一条直线与多边形相交超过两次，则这个多边形是非凸的。

法线（normal）

一个三个分量的平面方程，定义了一个平面或曲面的方向，但不是位置。

规一化（normalized）

用法线的每一个分量的平方和的平方根去除每一个分量。然后，如果法线认为是从原点到（nx', ny', nz'）的一个向量，则该向量为单位长度。其数学式为：

factor = sqrt(nx^2 + ny^2 + nz^2)

nx' = nx / factor

ny' = ny / factor

nz' = nz / factor

法线向量（normal vectors）

同法线。

非均匀有理 B 样条（NURBS）

非一致有理 B 样条。这是一个通用的办法，用来指定曲线或曲面的参数。参见附录 3 中的 GLU NURBS 程序。

O

物体（object）

指一种作为图元的集合来渲染的、物体坐标下的模型。

物体坐标（object coordinates）

在任何 OpenGL 转换之前的坐标系。

正交投影图（orthographic）

非透视的投影，像在一些工程绘图中，没有透视效果。

P

参数（parameters）

作为 OpenGL 命令的参数传递的数值。有些时候某个数值是通过引用 OpenGL 命令传递的。

透视分割（perspective division）

将 x, y 和 z 在裁剪坐标中进行分割。

像素（pixel）

指图形元素。在帧缓冲器中所有位平面的位置(x, y)处的位数组成了单像素(x, y)。在客户内存的映像中，一个像素是一个元素族。在 OpenGL 窗口坐标下，每一像素对应 1.0×1.0 屏幕面积。名为 x,y 的像素，其左下角坐标为(x, y)，而右上角坐标为(x+1, y+1)。

点（point）

空间的一个确切的位置，渲染时看作一个有限直径的点。

多边形（polygon）

一个近似平面的表面，该平面由顶点所确定的边围成。三角形网格中的每一个三角形是一个多边形，四边形网格中的四边形也是如此。由 glRect*( )定义的矩形也是一个多边形。

图元（primitive）

一个点，一条线，一个多边形，一幅位图或是一幅映像。注意：并不仅仅只是一个点，

一条线或一个多边形！

投影矩阵（projection matrix）

指将点、线、多边形和光栅位置从人眼坐标转换至裁剪坐标中的 4 阶方阵。

R

光栅化（rasterized）

由一个投影点、线、或多边形或位图的像素或映像转换成片原，每一个片原对应于帧缓冲器中的一个像素。注意所有的图元都被光栅化，而不仅仅是点、线和多边形。

渲染（rendering）

将在物体坐标系中定义的图元转换为帧缓存器中的映像。渲染是 OpenGL 中的基本操作。

RGBA

红色、绿色、蓝色和 Alpha。

RGBA 模式（RGBA mode）

如果一个 OpenGL 的内容表存储的是红色、绿色、蓝色和 alpah 颜色分量而不是颜色索引，则是处于 RGBA 模式。

S

服务器（server）

执行 OpenGL 命令的计算机。它可能不是命令发出的计算机。参见客户（client）。

阴影（shading）

光栅化过程中，在多边形的内部或是线的顶点之间进行颜色插值的过程。

单缓存（single-buffering）

没有背面颜色缓存的 OpenGL 的内容表称为单缓冲器。可以利用这些内容表来制作动画，但渲染时要注意避免视觉的干扰闪动。

奇异矩阵（singular matrix）

指一个不可逆矩阵。在几何上，奇异矩阵表示一种变换，这种变换将直线上的点沿一条直线叠并为一个单点。

点划（stipple）

一个一维或二维的二进制模式，避免生成值为零的片原。线的点划是一维的，相对于一条线的起点来使用。多边形点划是二维的，应用时相对窗口的方向固定。

T

网格化（tessellation）

是将一个解析表面的一部分简化为一个多边形网格，或是一条解析曲线的一部分简化为一系列的直线。

纹理单元（texel）

指一种纹理单元。这种纹理单元可以通过纹理内存获取，并表示应用于相应片原的纹理颜色。

纹理（textures）

一维或二维映像，用于修改光栅化产生的片原颜色。

纹理映射（texture mapping）

指将映像（纹理）用于图元的过程。纹理映射常用来为场景增加真实感。例如，可以将建筑物正面图像用到表示一垛墙的多边形上。

纹理矩阵（texture matrix）

4 阶方阵，将纹理坐标从指定的坐标变为用于插值及纹理查找的坐标。

变换（transformations）

指空间的转换。在 OpenGL 中，变换仅限于投影变换，包括可由一个 4 阶方阵表示的任何客体。投影变换包括旋转、平移、（非一致）沿坐标轴的比例变换、透视变换、以及多种变换的组合。

V

视点（viewpoint）

人眼坐标系或裁剪坐标系的原点，取决于内容表含意。例如，在讨论光照时，视点是人眼坐标系的原点。在讨论投影时，视点是裁剪坐标系的原点。在典型的投影矩阵中，人眼坐标系和裁剪坐标系的原点在相同的位置。

视图体（view volume）

裁剪坐标中的体，其坐标满足以下三个条件：

$$-w \leq x \leq w$$
$$-w \leq y \leq w$$
$$-w \leq z \leq w$$

该体积之外的几何图元将被裁剪掉。

$$-w \leq x \leq w$$
$$-w \leq y \leq w$$
$$-w \leq z \leq w$$

X

窗口（window）

帧缓冲区的子区域，通常为矩形，其所有像素具有相同的缓冲状态。一个 OpenGL 内容表在同一时刻只对同一窗口进行渲染。

窗口对齐（window-aligned）

在提及线段或多边形的边时，已经暗含指出了它们平行于窗口的边界（在 OpenGL 中，窗口是具有水平及垂直边界的矩形）。在涉及多边形时，暗含了该图案相对于窗口原点是固定的。

窗口坐标（window coordinates）

指窗口坐标系。它对于区别离散的像素值和连续的窗口坐标值非常重要。例如，一个窗口左下角的像素是（0，0）；而该像素中心的窗口坐标是（0.5，0.5，z)。注意：窗口坐标包括深度即 z 分量，该分量也是连续的。

线框（wireframe）

表示仅包含线段的物体。通常，线段指的是多边形的边。

X 窗口系统（X Window System）

运行 OpenGL 的大多数计算机所用到的窗口系统。GLX 是 OpenGL 对 X 窗口系统的扩展。

Document generated by Anna's Archive around 2023-2024 as part of the DuXiu collection
(https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):
{
  "filename": "MTM1MzkyOTkuemlw",
  "filename_decoded": "13539299.zip",
  "filesize": 77911393,
  "md5": "1faeb280e67223e16fe345b99019a70e",
  "header_md5": "613320043a91fdcac99550963e9dcbe7",
  "sha1": "794d520e3f588529c22824276fc320b37705590e7",
  "sha256": "ff18fa804a90135dd55718a810651a3e44d68482d5cc3a4a88dce7365792af2a",
  "crc32": 4137313973,
  "zip_password": "",
  "uncompressed_size": 98415754,
  "pdg_dir_name":
"OpenGL\u2550\u255d\u2568\u256c\u2502\u2560\u2568\u2265\u2554\u03a6\u255d\u255e\u2553\u2555\u2500\u2567_13539299",
  "pdg_main_pages_found": 460,
  "pdg_main_pages_max": 460,
  "total_pages": 486,
  "total_pixels": 2784468096,
  "pdf_generation_missing_pages": false
}